

AD-A154 825

SOFTWARE DESIGN PROTOTYPING USING ADA(U) NAVAL SURFACE  
WEAPONS CENTER DAHLGREN VA M W MASTERS ET AL. SEP 83  
NSWC/TR-82-417

1/1.

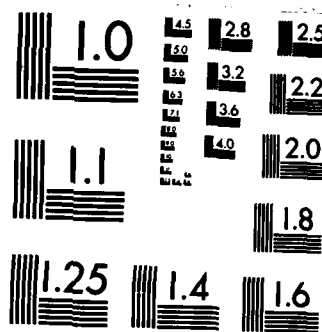
UNCLASSIFIED

F/G 9/2

NL

END

ENV



MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

ry

AD-A154 825

NSWC TR 82-417

## SOFTWARE DESIGN PROTOTYPING USING ADA

BY MICHAEL W. MASTERS

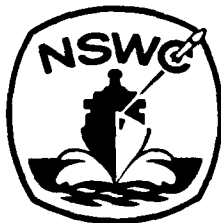
MICHAEL J. KUCHINSKI

COMBAT SYSTEMS DEPARTMENT

SEPTEMBER 1983

Approved for public release; distribution unlimited.

DTIC FILE COPY



**NAVAL SURFACE WEAPONS CENTER**

Dahlgren, Virginia 22448 • Silver Spring, Maryland 20910

DTIC  
ELECTE  
JUN 6 1985  
S D  
E

85 05 10 063

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM										
1. REPORT NUMBER NSWC TR 82-417	2. GOVT ACCESSION NO. AD-A154825	3. RECIPIENT'S CATALOG NUMBER										
4. TITLE (and Subtitle) SOFTWARE DESIGN PROTOTYPING USING ADA		5. TYPE OF REPORT & PERIOD COVERED Final										
7. AUTHOR(s) Michael W. Masters Michael J. Kuchinski		6. PERFORMING ORG. REPORT NUMBER										
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Surface Weapons Center (N21) Dahlgren, VA 22448		8. CONTRACT OR GRANT NUMBER(s)										
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Sea Systems Command Washington, DC 20362		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS SCN-funded										
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE September 1983										
		13. NUMBER OF PAGES 52										
		15. SECURITY CLASS. (of this report) UNCLASSIFIED										
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE										
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.												
<table border="1"> <thead> <tr> <th colspan="2">Accession For</th> </tr> </thead> <tbody> <tr> <td>NTIS GRA&amp;I</td> <td><input checked="" type="checkbox"/></td> </tr> <tr> <td>DTIC TAB</td> <td><input type="checkbox"/></td> </tr> <tr> <td>Unannounced</td> <td><input type="checkbox"/></td> </tr> <tr> <td>Justification</td> <td></td> </tr> </tbody> </table>			Accession For		NTIS GRA&I	<input checked="" type="checkbox"/>	DTIC TAB	<input type="checkbox"/>	Unannounced	<input type="checkbox"/>	Justification	
Accession For												
NTIS GRA&I	<input checked="" type="checkbox"/>											
DTIC TAB	<input type="checkbox"/>											
Unannounced	<input type="checkbox"/>											
Justification												
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)												
<table border="1"> <tbody> <tr> <td>By</td> <td></td> </tr> <tr> <td>Distribution/</td> <td></td> </tr> <tr> <td>Availability Codes</td> <td></td> </tr> </tbody> </table>			By		Distribution/		Availability Codes					
By												
Distribution/												
Availability Codes												
18. SUPPLEMENTARY NOTES												
<table border="1"> <tbody> <tr> <td>Dist.</td> <td>Avail and/or</td> </tr> <tr> <td>A/</td> <td>Special</td> </tr> </tbody> </table>			Dist.	Avail and/or	A/	Special						
Dist.	Avail and/or											
A/	Special											
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)												
<table border="0"> <tbody> <tr> <td>Ada</td> <td>Software Design</td> </tr> <tr> <td>High-Level Programming Language</td> <td>Software Development</td> </tr> <tr> <td>Program Design Language (PDL)</td> <td>Design Prototyping</td> </tr> <tr> <td>Computer Program</td> <td></td> </tr> </tbody> </table>			Ada	Software Design	High-Level Programming Language	Software Development	Program Design Language (PDL)	Design Prototyping	Computer Program			
Ada	Software Design											
High-Level Programming Language	Software Development											
Program Design Language (PDL)	Design Prototyping											
Computer Program												
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)												
<p>The investigation of the methodology for software design prototyping using Ada as a program design language (PDL) involves taking a system engineering approach to software development. A proposal is made to express design characteristics as Ada programs in an effort to provide executability of the design from its earliest specification. This approach is subsequently given more substance by an examination of the methodology from three distinct perspectives:</p>												

(over)

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE  
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)


UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (Continued)

- 1) The qualities of an "ideal" PDL are put forth and Ada is compared with this idealized PDL; Ada compares favorably with this model in most, if not all, areas.
- 2) The qualities of an "ideal" software prototype are specified, and the Ada prototype program is measured against this idealized prototype. Ada exhibits a number of characteristics that lend themselves well to the gradual refinement of a prototype program; it also shows itself to be highly supportive of testing and validation of the design as that design matures. It is found that a prototype developed in Ada, using this methodology, offers distinct advantages over the traditional software development. And
- 3) A step-by-step guide to the use of Ada as a PDL in a design prototyping environment is given.

*Additional benefits: high level programming languages;  
test beds; ACSIS (AEGIS Control System  
Interface Simulator),*



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

FOREWORD

In September 1981, a proposal was made to the NSWC IED panel to investigate a methodology for software design prototyping in the context of Navy tactical embedded computers and real-time software. The methodology is based on utilization of the Ada programming language as a program design language. Funding was granted for one man-year of effort spanning 1 October 1981 through 30 September 1982.

This report was reviewed by Robert J. Crowder, Head, Systems Evaluation and Control Branch; Daniel Green, Staff Scientist; and R. N. Cain, Head, AEGIS Ship Combat Systems Division.

Released by:

  
THOMAS A. CLARE, Head  
Combat Systems Department

## PREFACE

The authors would like to provide the reader with some background information regarding the subject of this report. They attempt to develop and describe a methodology for software design and development. In order to give substance to the ideas treated herein, the study was performed within the context of a trial conceptual software design effort. An early version of an Ada compiler was used in the construction of the design. The test bed chosen for the study was the AEGIS Combat System Interface Simulator (ACSIS) for the AEGIS Ship Combat System. Briefly, AEGIS is a computer-controlled, integrated ship weapon system, designed to detect, track, and engage incoming missiles. ACSIS is slated to be the simulator program by which the AEGIS computer programs are maintained throughout their lifespan. The choice of ACSIS was made due to the fact that it is destined to be a real-time program, and such an environment is well suited to the methodology.

The Ada compiler used in this study was developed by Telesoft. The computer used was an Intellimac IN 7000D microcomputer. The Ada compiler was, unfortunately, incomplete. Consequently, some of the techniques described in this report are inevitably implemented differently than they would have been had a full compiler been available. In some cases techniques are proposed and described that could not be compiled and tested. Such examples are at present unverified. However, every attempt has been made to ensure their accuracy.

Many of the Ada code segments given as examples in this report are excerpts taken from the test bed program. However, they represent only a small part of the program produced from the trial design effort.

## CONTENTS

	<u>Page</u>
INTRODUCTION . . . . .	1
BACKGROUND . . . . .	1
DESIGN PROTOTYPING METHODOLOGY . . . . .	3
PROGRAM DESIGN LANGUAGES . . . . .	7
IDEALIZED PDL . . . . .	7
ANALYSIS OF ADA AS A PDL . . . . .	8
SUMMARY . . . . .	19
PROTOTYPE COMPUTER PROGRAMS . . . . .	19
IDEALIZED PROTOTYPE . . . . .	19
DESIGN PROTOTYPING VS CONVENTIONAL SOFTWARE DEVELOPMENT . . . . .	20
SUMMARY . . . . .	31
APPLICATION OF DESIGN PROTOTYPING METHODOLOGY . . . . .	31
APPLICATION ENVIRONMENT . . . . .	31
PHASE ONE OF PHYSICAL DESIGN . . . . .	31
PHASE TWO OF PHYSICAL DESIGN . . . . .	37
PHASES THREE AND FOUR OF PHYSICAL DESIGN . . . . .	37
CONCLUSIONS . . . . .	40
STATE OF THE ART . . . . .	40
SUMMARY . . . . .	40
POSTSCRIPT . . . . .	40
REFERENCES . . . . .	43
DISTRIBUTION . . . . .	(1)

## ILLUSTRATIONS

<u>Figure</u>		<u>Page</u>
1	TRADITIONAL SOFTWARE DEVELOPMENT PROCESS . . . . .	1
2	SYSTEM ENGINEERING PROCESS . . . . .	2
3	PROTOTYPE DEVELOPMENT CYCLE . . . . .	4
4	SAMPLE DATA PACKAGE FOR TRACK-RELATED MODULES . . . . .	9
5	ADA PACKAGE DECLARATION (VISIBLE PORTION) WITH PRIVATE TYPES . . . . .	11
6	ADA PACKAGE BODY (INVISIBLE PORTION) . . . . .	12
7	STUBBED-OUT ADA SUBPROGRAMS . . . . .	12
8	TYPICAL ADA TASK . . . . .	13
9	STEP-WISE REFINEMENT OF ADA SUBPROGRAM . . . . .	15
10	ADA STRUCTURES THAT SUPPORT SEQUENCING LOGIC . . . . .	17
11	CPU TIME CONSUMPTION EXAMPLE . . . . .	22
12	ADA SEPARATE FEATURE . . . . .	24
13	ADA TASK SYNCHRONIZATION MECHANISM . . . . .	25
14	INTER-TASK MAILBOX COMMUNICATION STRUCTURE . . . . .	28
15	ADA SHARED RESOURCE EXAMPLE . . . . .	29
16	TYPICAL TASK DEVELOPMENT . . . . .	32
17	SYSTEM MODULE ARCHITECTURE DIAGRAM . . . . .	33
18	SYSTEM MODULE ARCHITECTURE IN CODE FORM . . . . .	34
19	VISIBILITY HIERARCHY FROM THE COMPILER'S VIEWPOINT . . . . .	35
20	STUBBED-OUT MESSAGE STRUCTURE . . . . .	36
21	TYPICAL SUBPROGRAM PACKAGE AT END OF PHASE ONE . . . . .	36
22	TYPICAL SUBPROGRAM PACKAGE DURING PHASE TWO . . . . .	38

## TABLES

<u>Table</u>		<u>Page</u>
1	PHYSICAL DESIGN VS FUNCTIONAL REQUIREMENTS MATRIX . . . . .	21
2	COMPARISON OF ADA, ATES, AND VMS . . . . .	27

## INTRODUCTION

## BACKGROUND

The software development process is frequently characterized as including several basic stages:<sup>1</sup> requirements definition, specification, design, code and debug, and test and integration (Figure 1). There are difficulties inherent in this traditional approach, however. The most significant problem involves errors in the requirements and/or the design. These errors may not be discovered until the code and debug phase or test and integration phase. When errors go undetected for so long into the development process, costly corrections ensue.

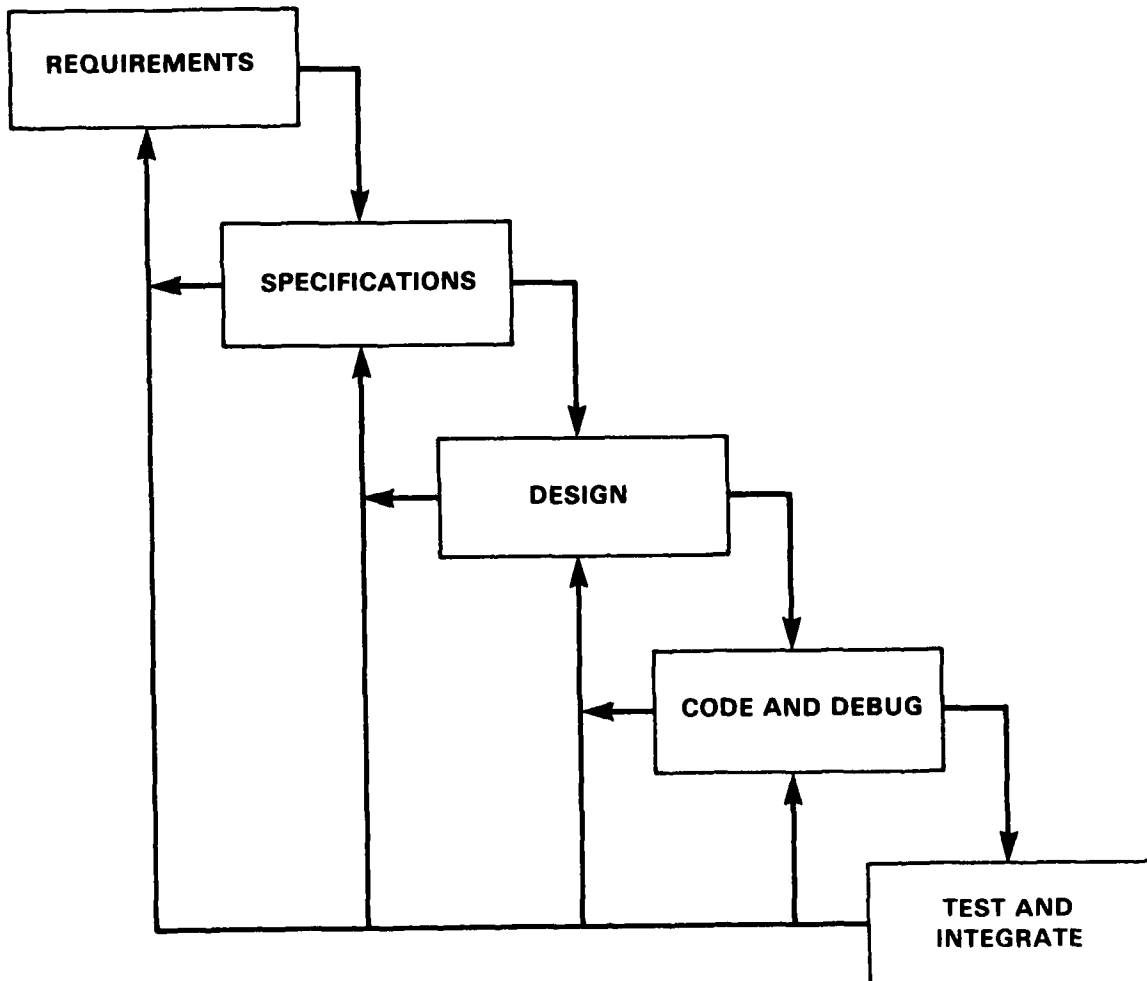


FIGURE 1. TRADITIONAL SOFTWARE DEVELOPMENT PROCESS

An approach commonly used in hardware development, the system engineering process (Figure 2), identifies five developmental stages:<sup>2</sup> requirements, function analysis, synthesis, evaluation and decision, and system specification. The functional analysis stage involves the study of the system's logical functional breakdown for the purpose of physically allocating those functions. Synthesis essentially entails a trial system design and the development of a working model or prototype. Evaluation has to do with the verification of the design via prototype, with the provision for feedback to the design and requirements phases. System specification precedes full production implementation. The system specification for the anticipated production of a great number of hardware units is somewhat different from the situation of software system specification. Nevertheless, the authors contend that this very same approach, so common in the development of hardware, can also be used in software development.

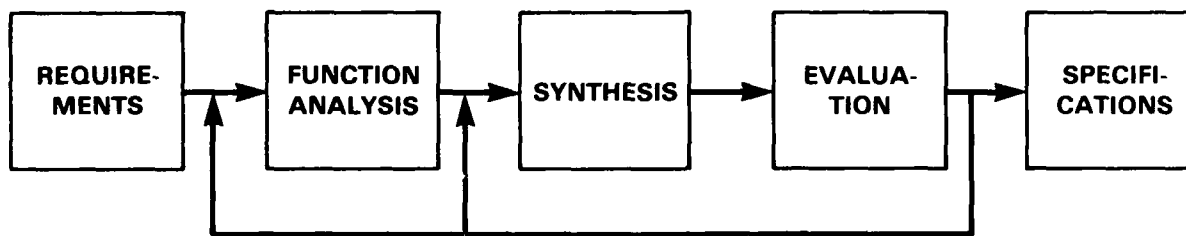


FIGURE 2. SYSTEM ENGINEERING PROCESS

The notion of computer program prototyping is one that has not attained popularity in the field of computer science. Nevertheless, it can provide the same type of benefits as hardware prototyping. Loosely speaking, a prototype is a computer program that is largely diagnostic and provides a means by which errors of design can be detected, or the absence thereof can be verified.<sup>3</sup> In this way, a computer program's design can be examined and tested to see if it accurately reflects the specified requirements.

In the current life-cycle models, the validation of the design depends upon the implementation of that design into a production computer program. This production program constitutes the first time that the design is scrutinized from the standpoint of executability. Cost and schedule overruns and inadequate performance may indicate a faulty design and, as such, they may lead to a substantial amount of "backtracking" in the design process. A possible need for backtracking suggests that the ability to validate the executability of the design and the ability to determine user satisfaction early in the design process can be highly desirable. To this end, prototype development has recently begun to appear as a distinct phase in some software life-cycle models.<sup>1,4</sup>

A technique for specifying the program's design must be determined. A number of specification tools are available to expedite the computer program design process; the most common of these is the flowchart. A more recently developed tool would be any of the various program design languages (PDLs). An automated means of translation from PDL to code has also been considered desirable;<sup>5</sup> although, the idea of prototyping in this context seems to have gone largely unnoticed.

In response, the authors have undertaken to use a high-level programming language (Ada, the new Department of Defense computer program language) in much the same way that one would use a PDL. The significant advantage to this approach is that by adhering to a prescribed set of conventions, the program's design can be subjected to verification from its earliest stages and, indeed, throughout its development. Thus, upon conclusion of the design specification phase, a validated and verified prototype program already exists; the two are largely one and the same.

The above-mentioned approach gives rise to a "software-first" methodology, in which a computer program is developed incrementally. Each increment corresponds to a phase in the design development and is the means by which the design is specified. With this approach, each of the intermediate increments is considered to be a prototype of increasing complexity.

This methodology is one that could ostensibly be used with any high-level computer language. Nevertheless, the authors feel that best results can be achieved through the use of a block-structured, English-like programming language such as Pascal or Ada. Given the broad range of semantic capabilities in Ada and the attention that has been devoted to readability and clarity, the authors have chosen to propose the methodology for use in conjunction with Ada. The development of the underlying ideas has been sparked by the requirements for software design and development in a real-time, military embedded computer system environment. However, the methodology is not limited to military applications and could be used in a wide variety of programming application environments. The application of the methodology to non-real-time applications has not been seriously considered at this time.

#### DESIGN PROTOTYPING METHODOLOGY

The methodology described herein and the philosophy behind it can be briefly summarized in the following manner. First and foremost, the design is developed in a top-down fashion. High-level aspects of the design, including preliminary resource consumption information, are put forth in the form of a computer program (albeit a rudimentary one) and subsequent levels of detail are added incrementally. (A description of the techniques used to handle unspecified components of the design will be dealt with later in this report.) Each time a major milestone is reached in the design development, the program is compiled and run in order to ensure executability of the design. No new design components are added until successful compilation and execution are achieved. (The reader is cautioned against confusing this methodology with the old "code-now, design-later" philosophy.<sup>4</sup> The authors' methodology is a structured and orderly process and is completely compatible with the system engineering process.) With this methodology, the high-level language becomes the medium by which the design is expressed, while the executability factor provides an additional advantage beyond standard design techniques.

Another significant aspect of the design philosophy concerns the authors' pre-  
 assumptions about the functional requirements and physical design. The "logical" part of the design involves the breakdown of that design into functional areas; that is, the systematic grouping of different design components according to the functions that they perform. Such a breakdown generally falls into the category of requirements definition. The authors presume the existence of a set of well-defined requirements and functional relationships as a necessary precursor to the implementation of the methodology. In addition, an operating system with support routines, a

hardware suite, and requirements-oriented scenarios (i.e., suitable test data) are needed as a backdrop to the methodology. Physical design constitutes the implementation of the functional areas into physical components. The methodology described herein effectively provides for the development of the physical design.

The key element of the prototype process described in this report is incremental design and development, a concept which has in the past been applied successfully to large-scale system developments.<sup>6</sup> In this approach, the development of the physical design is broken down into four distinct stages (Figure 3). First, the inter-task communication structure and the system-level architecture are specified. In Ada, of course, the interface to the run-time support environment is defined as a part of the language. System-level architecture involves the design of archetypical processing units or modules and identification of their interrelationships.

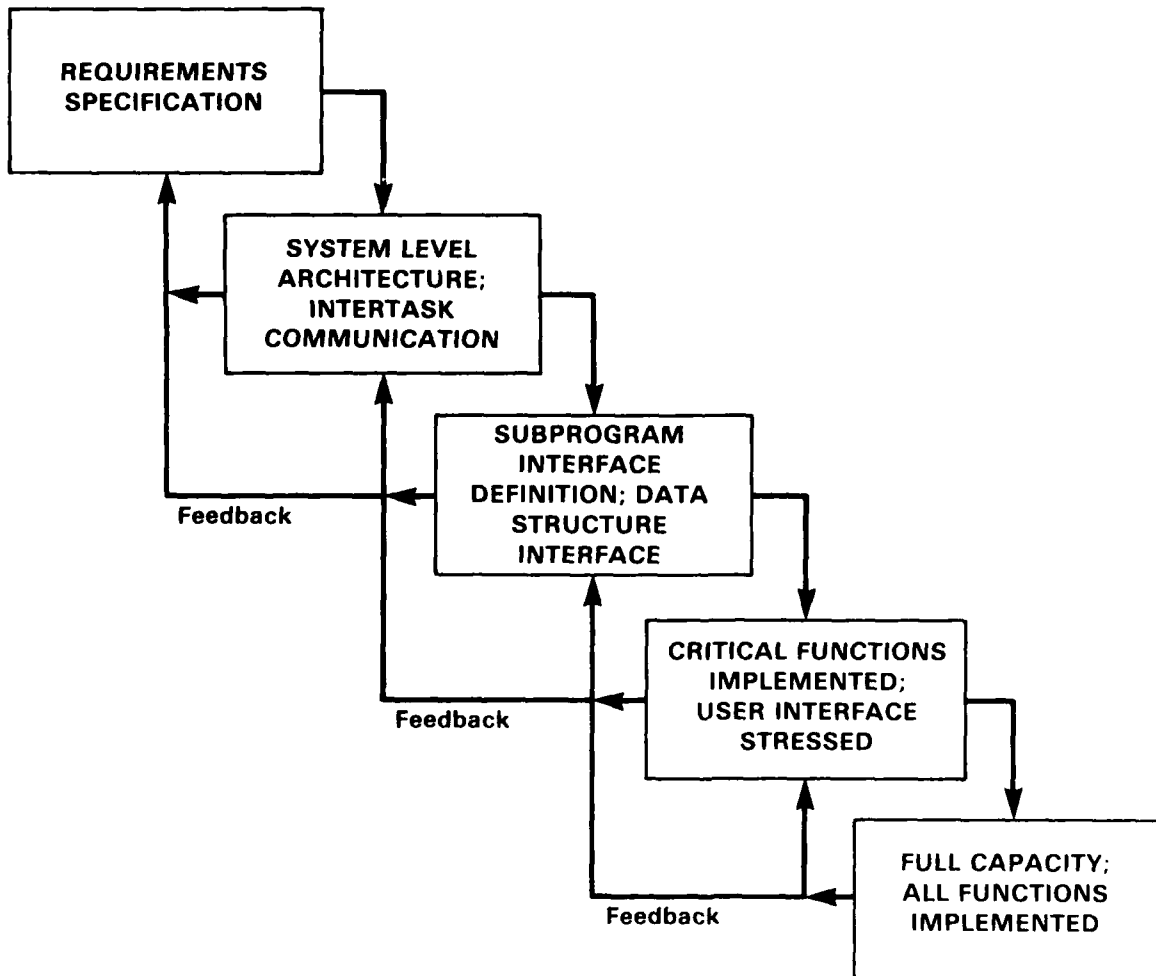


FIGURE 3. PROTOTYPE DEVELOPMENT CYCLE

At this stage of development, major processing requirements are allocated to subprograms; entries and associated processing segments are defined; and global data structures and intertask data communications requirements are identified in a conceptual manner. For example, records may be identified, but components not completely defined. This provides a high-level framework upon which the components of the program can be built. Resource allocation can even enter into the system's configuration at this early stage, with the resources themselves being based on estimates of anticipated system performance. Although the computer program that results from this level of design has little of the intended functionality of the final system, it does, nevertheless, when compiled and executed, permit early evaluation of resource consumption and control flow at the system module or task level.

Next, the subprogram structure and supporting data structures are defined. At this stage comes establishment of detailed control and interface design for subprograms and data. This step constitutes specification of primary and supporting subprograms, as well as implementation of how these subprograms interact with the program as a whole.

Note that in both of the first two stages, execution of the program requires that conditionals be resolved in order to permit control flow to proceed. In the absence of program functionality to allow these choices to be made dynamically, this amounts to forcing program execution to take a more or less predetermined path. While this mode of testing cannot fully substitute for a completely functional mode of operation, it does permit evaluation of resource consumption and flow of control throughout the program. In other words, the first two stages of development are simulations meant to test control flow and resource consumption at a fairly high level of detail. These stages will be dealt with in more detail later.

In the third stage, critical functions are implemented with an emphasis on the user interface. Here, the value of early user interaction becomes manifest. Early feedback from the user ensures greater user satisfaction with the delivered system. The reader may find it easier to conceptualize this stage of the development as emulating a degraded version of the completed prototype.

The completion of the fourth and final step constitutes completion of the prototype, with all functions implemented in their entirety. Depending upon the degree of rigor with which the program designer applies the methodology, the prototype may be only "functionally equivalent" to the finished system. That is, the prototype's algorithms, while complete, are perhaps more crudely implemented than they would be in the finished system. It should be reemphasized that verification of the prototype has taken place throughout its development. Upon completion of this final stage, a validated prototype exists that can be used as a springboard for full-scale development of the system.

The four-stage process just described emphasizes the similarities between hardware development and software prototyping. However, it is possible to reinterpret the fourth and possibly the third stage of the process so that the product that emerges at the end of the process is not a throwaway but actually is the final system. Thus the validated program structure developed during stages one and two forms part of the final system. In addition, during stages three and four, actual developed algorithms, as opposed to crudely implemented, functionally equivalent algorithms, are developed.

ess has already been successfully developed at the Naval Surface Weapons Center using PSL/PSA with the CMS-2 military programming language.<sup>17</sup> CMS-2 source and object files are used as input, a PSL/PSA data base is produced, and documentation reports are subsequently generated. A similar process has been developed by the University of Michigan ISDOS project to build PSL/PSA data bases from Extended TRAN code.<sup>18</sup>

### DL Should Support Early Prototyping

The ability to translate a design into an executable prototype program early on in the design phase is a worthwhile quality but not one that is addressed by traditional PDLs. Early prototyping ensures against flaws in the design going unnoticed until late in the design development. Using what is commonly known as the "skeleton and stub" approach, early phases of the design can be put forth as executable programs in whatever high-order language is being used. Admittedly, these programs do not do very much from the standpoint of actual processing, but trace statements such as "Update of all Great Circle tracks now in progress" can be used for design verification until full algorithms are implemented. In the authors' experience, the Ada programming language has shown itself to be extremely well suited to this technique.

### MARY

Ada matches closely the requirements of an ideal PDL. Like most existing PDLs, its support for automated documentation is weak. Unlike most other PDLs, it incorporates a prototyping capability. In light of these findings, the authors roughly endorse the use of Ada itself as a PDL.

## PROTOTYPE COMPUTER PROGRAMS

### ALIZED PROTOTYPE

With regard to the qualities that one would expect to find in a prototype computer program, the authors use a definition different from some of the more commonly held viewpoints. A prototype need not be, as some would maintain, a "quick and dirty" attempt to translate the design into executable code. Rather, the prototype should be a more sophisticated representation and can incorporate (albeit in skeletal form) any portion of the design that is destined to be included in the final system. It is the prototype necessarily a "throw-away." This is especially true when using a methodology that ensures a high degree of correlation between the prototype and the design specification. Finally, inefficiency is not a quality to be tolerated in a prototype computer program. In other more typical prototypes, the emphasis is on "functional completeness," however crude the implementations. In a prototype developed with the authors' methodology, the emphasis is on "design completeness" in architecture and resource allocation validity. (The reader who is skeptical of these remarks is asked to set aside any preferential views on the subject and judge the methodology in the context of the authors' definition of prototyping.)

characterized by the conditional entry call statement of Ada, again nested within a loop structure. Unfortunately, the authors have not been able to make full use of this active/passive philosophy due to the fact that their preliminary Ada compiler, the Telesoft implementation,<sup>16</sup> did not include the conditional entry call.

#### A PDL Should be Functionally Similar to the Implementation Language

It is reasonable to assume that if a high-level language is used as a PDL, then the program designer intends to use the same high-level language as the implementation language. Therefore, not only is the design language functionally similar to the implementation language, but the two are in fact the same.

#### A PDL Should Have Unexacting Syntax Constructs

One of the most frequently heard criticisms regarding Ada is the size and complexity of its syntax; the authors concede that Ada does have a great many syntax rules. In light of this, the program designer should examine the syntax critically to determine which rules are relevant for a given application and which ones should go unused. In other words, the very size of Ada enables the user to tailor the language to a particular situation through the selective elimination of certain syntax rules. Implicit here is the ability to make the outer syntax of the language as simple or complex as one wishes. [The authors foresee possible tools as part of the Ada Program Support Environment (APSE) that would be able to identify a specified Ada subset for a particular project. At the compiler level, subsetting in Ada is not permitted.] Concerning inner syntax, comment entries and long identifier names can be used in lieu of free-form English. This is a slightly more restrictive inner syntax than most PDLs offer. However, this drawback is offset by the benefits of an exhaustive identification of conditions and computations that must be elaborated later in the design.

#### A PDL Should Support Validation and Verification

What better way to verify a design specification than to have a working prototype at any stage in the design development process. As was stated earlier, the ultimate test of a program design is the ability to express that design as executable code. This is, by far, the fundamental advantage of using a high-level language as a PDL. Future efforts combining program correctness techniques with executable PDLs would appear to be extremely appropriate.

#### A PDL Should Support the Generation of Documentation

Here is one area in which Ada (and, for that matter, virtually any high-level programming language) falls short of the idealized PDL. However, since Ada (and other high-level languages) are machine processable, the authors envision some possible ways of alleviating this shortcoming. One possible approach might be the development of some support computer programs (perhaps as a part of the APSE) that could search through Ada code listings and produce input files for some data storage and analysis language [e.g., the Problem Statement Language/Problem Statement Analyzer (PSL/PSA)] and thereby produce various forms of documentation. Such a

**Sample Ada Timed Entry Call Statement:**

```

select
  OPERATOR_CONSOLE_SUPPORT.RECEIVE_MESSAGE (THIS_MESSAGE);
  -- Perform a rendezvous if one is possible within 30 seconds
or
  delay 30.0;
  -- Operator Console Support busy .... do something else
end select;

```

**Sample ADA Conditional Entry Call Statement And Exception Statement:**

```

select
  MAILBOX.SEND_MESSAGE (THIS_MESSAGE);
else
  if LENGTHQ (MESSAGE_QUEUE) > MAXIMUM_MESSAGE_COUNT then
    raise MESSAGE_QUEUE_FAULT;
  end if;
end select;

```

**Binary Semaphore Construct Using The Ada Selective Wait Statement:**

```

task SEMAPHORE is
  entry LOCK;
  entry UNLOCK;
end SEMAPHORE;

task body SEMAPHORE is
  LOCKED: BOOLEAN := false;
begin
  loop
    select
      when not LOCKED =>
        accept LOCK do
          LOCKED := true;
        end;
      or
        when LOCKED =>
          accept UNLOCK do
            LOCKED := false;
          end;
      or
        terminate;
    end select;
  end loop;
end SEMAPHORE;

```

FIGURE 10. ADA STRUCTURES THAT SUPPORT SEQUENCING LOGIC

```

with TRACK_FILE_PACKAGE,
    TRIGONOMETRY_SUBPROGRAM_PACKAGE;

procedure LOXODROME_UPDATE is
    ARC_LENGTH: FLOAT;
    RADIUS: FLOAT;
    OLD_LATITUDE: FLOAT;
    RADIUS_OF_EARTH: FLOAT := 2.0856E07;
    PI: FLOAT := 3.14159;
begin
    -- Traverse the Track File and examine the Path field of each active
    -- track to see if it is a Loxodrome track....if so, update the
    -- track's record as per "The VNR Concise Encyclopedia of Mathematics"
    -- by Gellert, et. al.16
    for INDEX in 1..100 loop
        if TRACK_FILE(INDEX).ACTIVE and
            TRACK_FILE(INDEX).PATH = LOXODROME then
            ARC_LENGTH := TRACK_FILE(INDEX).SPEED * TIME;
            RADIUS := RADIUS_OF_EARTH + TRACK_FILE(INDEX).ALTITUDE;
            OLD_LATITUDE := TRACK_FILE(INDEX).LATITUDE;
            TRACK_FILE(INDEX).LATITUDE :=
                TRACK_FILE(INDEX).LATITUDE +
                (ARC_LENGTH * COS(TRACK_FILE(INDEX).HEADING) / RADIUS);
            TRACK_FILE(INDEX).LONGITUDE :=
                TRACK_FILE(INDEX).LONGITUDE +
                TAN(TRACK_FILE(INDEX).HEADING) *
                ((LN TAN(PI/4.0 + TRACK_FILE(INDEX).LATITUDE/2.0)) -
                 (LN TAN(PI/4.0 + OLD_LATITUDE/2.0)));
        end if;
    end loop;
end LOXODROME_UPDATE;

```

FIGURE 9. STEP-WISE REFINEMENT OF AN ADA SUBPROGRAM (Continued)

```

with TRACK_FILE_PACKAGE;
procedure LOXODROME_UPDATE is
begin
  STALL (0.1); -- time in seconds assumed
  -- Traverse the Track File and examine the Path field of each active
  -- track to see if it is a Loxodrome track....if so, update the
  -- track's record as per "The VNR Concise Encyclopedia of Mathematics"
  -- by Gellert, et. al.15
end LOXODROME_UPDATE;

with TRACK_FILE_PACKAGE;
procedure LOXODROME_UPDATE is
begin
  -- Traverse the Track File and examine the Path field of each active
  -- track to see if it is a Loxodrome track....if so, update the
  -- track's record as per "The VNR Concise Encyclopedia of Mathematics"
  -- by Gellert, et. al.15
  for INDEX in 1..100 loop
    if TRACK_FILE(INDEX).ACTIVE and
       TRACK_FILE(INDEX).PATH = LOXODROME then
      STALL (0.001); -- time in seconds assumed
      -- Arc Length = Speed * Time
      -- Radius = Radius of Earth + Altitude
      -- New Latitude = Latitude + (Arc Length * cos(Heading) / Radius)
      -- New Longitude = Longitude + tan(Heading * ((ln tan(Pi/4 +
      -- New Latitude/2)) - (ln tan(Pi/4 + Latitude/2)))
    end if;
  end loop;
end LOXODROME_UPDATE;

```

FIGURE 9. STEP-WISE REFINEMENT OF AN ADA SUBPROGRAM

A PDL Should Support Top-Down Design and Step-Wise Refinement

Part and parcel of the top-down design philosophy is the notion that details of the design should be developed in an evolutionary, step-wise fashion. While step-wise refinement can be used in program design environments other than top-down (e.g., bottom-up), the selection of top-down as the design approach necessitates the use of step-wise refinement. Early design efforts are lacking in the area of specific details, while later phases of the design possess ever-increasing amounts of detail until the entire design is specified in the final phase. The initial stub contains little or no information on how the subprogram will ultimately do its processing. However, subsequent versions of the subprogram contain increasing amounts of detail until, finally, the subprogram's algorithm is complete.

The way in which a high-level language can be used in step-wise refinement of the design can be seen from the following generalized example. A subprogram is proposed as an addition to the already existing design, but specification of the subprogram's processing is not yet feasible or practical. Instead, the subprogram is implemented in the form of a mere stub. Later in the design process, the subprogram's processing is "sketched out" in standard English and/or mathematical symbol-ogy contained in comment entries. Finally, in completing the prototype, the subprogram's processing is specified in executable algorithmic form. A specific example, expressed in Ada, can be seen in Figure 9. A high-level language (especially one like Ada) used in such a manner helps to support the notion of top-down system design. It should also be mentioned that the separate compilation capability of Ada makes it possible to integrate data and subprogram packages at different levels of detail in the design phase.

A PDL Should Support Low-Level and High-Level Sequencing Logic

The logical sequence in which events occur can be easily expressed through a number of built-in Ada features. Low-level sequencing logic can be handled by the Ada if statement, the loop statement, or the case statement. High-level sequencing logic, on the other hand, is dealt with using more sophisticated Ada structures such as the entry call, the timed entry call, the conditional entry call, and the selective wait statement (Figure 10). The entry call initiates a rendezvous and the timed entry call enables the initiation of a rendezvous at any point within a specified time interval. The rendezvous mechanism in Ada is useful in handling the synchronization of tasks. Asynchronous task communication, on the other hand, can be facilitated by the inclusion of an intermediary "mailbox" task. For example, if Task A wishes to send a message to Task B asynchronously, then Task A performs a rendezvous with the mailbox task that places the message in a queue. Task A then continues processing. The mailbox task performs a rendezvous with Task B once the message reaches the top of the queue. (The necessity of creating this mailbox task to effect asynchronous communication is a controversial aspect of the rendezvous mechanism. However, efficient implementations have been shown to be possible.<sup>14</sup>) Associated with this asynchronous capability is a feature that has been called the "active/passive" task structure.<sup>10</sup> Simply put, certain tasks can be designated as passive tasks in that they render a service to other tasks. (The mailbox task can be regarded as such.) They are generally characterized by their use of the Ada selective wait statement nested within a loop [e.g., the implementation of semaphores (Figure 10)]. Other tasks are referred to as active because they perform a fundamental unit of processing within the system. These latter tasks are usually

```

task body GLOBAL_COORDINATE_GENERATION is
  INTERVAL: constant DURATION: = 0.01;
  NEXT_TIME: CALENDAR.TIME;
  -- CALENDAR here refers to a predefined library package,
  -- whose function CLOCK returns the current value of TIME
  -----
  -- Other local data declarations
  .
  .
  .
begin
  accept COMMENCE do
    THIS_TASK.ID: = 4;
    THIS_TASK.NAME: = "Global Coordinate Generation";
    -- Other initialization routines
    .
    .
    .
    NEXT_TIME: = CALENDAR.CLOCK( ) + INTERVAL;
  end;
  loop
    select
      accept RECEIVE_MESSAGE (MESSAGE: in MESSAGE_TYPE) do
        ASSIMILATE_NEW_INFORMATION;
        -- Assimilate new information into data base
        end;
      or
        -- accept other entries and perform processing
        .
        .
        .
      or
        delay NEXT_TIME -- CALENDAR.CLOCK( );
        PERFORM_PERIODIC_UPDATE;
        -- Perform periodic update routines
        NEXT_TIME: = NEXT_TIME + INTERVAL;
        -- Generate message(s) to be sent to
        -- the Mailbox for subsequent transmittal
        -- to its(their) recipient(s)
        MAILBOX.SEND_MESSAGE (THIS_MESSAGE);
      else
        null;
      end select;
    exit when END_OF_PROGRAM;
  end loop;
end GLOBAL_COORDINATE_GENERATION;

```

FIGURE 8. A TYPICAL ADA TASK

```

package body MESSAGE QUEUE is
  procedure ADDQ (A QUEUE: in out QUEUE;
    MESSAGE ITEM: in MESSAGE) is
    -- This procedure adds a message to
    -- the end of the queue
    TEMPORARY: POINTER;
  begin
    TEMPORARY := new NODE(MESSAGE ITEM, null);
    if A QUEUE.HEAD = NULL THEN
      A QUEUE.HEAD := TEMPORARY;
      A QUEUE.TAIL := TEMPORARY;
    else
      A QUEUE.TAIL.ITEM POINTER := TEMPORARY;
      A QUEUE.TAIL := A QUEUE.TAIL.ITEM POINTER;
    end if;
    A QUEUE.COUNT := A QUEUE.COUNT + 1;
  end ADDQ;
  .
  .
  .
end MESSAGE QUEUE;

```

FIGURE 6. ADA PACKAGE BODY (INVISIBLE PORTION)

```

procedure GREAT_CIRCLE_UPDATE is
begin
  -- Update all Great Circle track records as per
  -- "American Practical Navigator"
  -- by Nathaniel Bowditch13
  null;
end GREAT_CIRCLE_UPDATE;

procedure SHIP_MOTION_GENERATOR_INTERFACE is
begin
  -- Assimilate ship motion information
  null;
end SHIP_MOTION_GENERATOR_INTERFACE;

procedure TRACK_REPORT_FROM_SPY_RADAR is
begin
  -- When completed, the program will at this time display to the
  -- operator a list of all targets that are currently being
  -- tracked by the SPY Radar System
  null;
end TRACK_REPORT_FROM_SPY_RADAR;

```

FIGURE 7. STUBBED-OUT ADA SUBPROGRAMS

```

package MESSAGE__QUEUE is
  type QUEUE is private;
  type MESSAGE is
    record
      ID: INTEGER;
      SOURCE: TASK__NAME;
      DESTINATION: TASK__NAME;
      -- Content field....
      -- to be decoded at destination
    end record;

  procedure ADDQ (A__QUEUE: in out QUEUE;
    MESSAGE__ITEM: in MESSAGE);
  procedure DELETEQ (A__QUEUE: in out QUEUE;
    MESSAGE__ITEM: out MESSAGE);
  function FRONTQ (A__QUEUE: in QUEUE) return MESSAGE;
  function IS__EMPTYQ (A__QUEUE: in QUEUE) return BOOLEAN;
  function IS__MEMBERQ (A__QUEUE: in QUEUE;
    MESSAGE__ITEM: in MESSAGE) return BOOLEAN;
  function LENGTHQ (A__QUEUE: in QUEUE) return INTEGER;

  private
    type NODE;
    type POINTER is access NODE;
    type NODE is
      record
        ITEM: MESSAGE;
        ITEM__POINTER: POINTER := null;
      end record;
    type QUEUE (MAXIMUM__COUNT: INTEGER) is
      record
        HEAD: POINTER := null;
        TAIL: POINTER := null;
        COUNT: INTEGER := 0;
      end record;
  end MESSAGE__QUEUE;

```

FIGURE 5. ADA PACKAGE DECLARATION (VISIBLE PORTION) WITH PRIVATE TYPES

A PDL Should Support Detail-Hiding and Data Abstraction

A good example of detail-hiding is the case of a frequently used subprogram. The issue of when this subprogram should be invoked and what processing it performs is independent of how the subprogram's processing is implemented. Thus, the details of how the subprogram performs its processing can and should be invisible to those portions of the system concerned with control flow. (Almost all high-level programming languages provide for this kind of detail-hiding.) Ada's visibility rules lend themselves well to this hiding of details and permit the declaration of functions and procedures to be visible, while their bodies remain hidden (Figure 5). In the area of data abstraction, Ada permits the declaration of private data types, the specifications of which are invisible to the user (see Figure 5). The package structure is the feature of Ada that permits subprograms and/or data structures to be logically separated from the program units that utilize them (Figure 6). The package permits encapsulation of data types. Data objects of the defined type may then be declared and manipulated using the appropriate operations. The user interface to subprograms and data is made visible in the package specification, while implementation details are hidden in the package body.

A PDL Should Permit Expression of Unspecified Parts of the Design

Comment entries and descriptive subprogram calls are instrumental in outlining unspecified portions of the design. English phrases and sentences are used in lieu of actual processing algorithms. Subprograms are "stubbed out" [i.e., are specified, but contain no true processing (Figure 7)]. (Lengthly Ada identifiers also prove useful in this regard.) If it is necessary to simulate central processing unit (CPU) time consumption by an algorithm where only a stub presently exists, a subprogram designed to stall for a period of time may be employed. An example will be given in the discussion of top-down design and step-wise refinement later in this section.

A PDL Should Permit Expression of the Physical Software System Architecture

The Ada programming language is particularly well suited to this area, not only with regard to control flow architecture, but with regard to data structures as well. The Ada task construct is useful in separating the various modular components of a system (Figure 8) when these modular components represent portions of the program that are intended to run concurrently. The rendezvous concept in Ada is the principal mode of communication between two or more Ada tasks and, as such, is exceedingly useful in expressing the communication protocol among modules. Concerning the expression of data structures, the Ada package construct makes it easy to separate data dedicated to certain functional areas from other data items. Ada visibility rules also play an important role in expressing system architecture. These results help to delineate those data items and subprograms that are the exclusive domain of a particular module from those that are globally accessible. (The development of the physical design will be dealt with in more detail later in this report.)

```

package TRACK_FILE_PACKAGE is
  type TRACK_PATH_TYPE is
    (GREAT_CIRCLE, LOXODROME);
  type TRACK_RECORD is
    record
      ACTIVE: BOOLEAN;
      ID: INTEGER;
      PATH: TRACK_PATH_TYPE;
      LATITUDE: FLOAT;
      LONGITUDE: FLOAT;
      HEADING: FLOAT;
      ALTITUDE: FLOAT;
      PITCH: FLOAT;
      SPEED: FLOAT;
    end record;
  TRACK_FILE: array (1..100)
    of TRACK_RECORD;
  type MANEUVER_TYPE is (NEW_SPEED,
    NEW_ALTITUDE, NEW_PITCH,
    NEW_HEADING);
  type MANEUVER_SPECIFICATION is
    record
      TRACK_ID: INTEGER;
      ACTION_TO_BE_TAKEN: MANEUVER_TYPE;
      GOAL_TO_BE_ATTAINED: FLOAT;
    end record;
end TRACK_FILE_PACKAGE;

with TRACK_FILE_PACKAGE;
task body GLOBAL_COORDINATE_GENERATION is
begin
  .
  .
  .
end GLOBAL_COORDINATE_GENERATION;

```

FIGURE 4. SAMPLE DATA PACKAGE FOR TRACK-RELATED MODULES

12. Support the generation of documentation
13. Support early prototyping

#### ANALYSIS OF ADA AS A PDL

In order to pursue an evaluation of Ada as a design language, each of the "ideal PDL" criteria will be examined and an assessment of Ada's performance in each area will be made.

##### A PDL Should Be Readable

One must keep in mind the fact that a design specification is primarily intended to communicate ideas to people. In light of this, a PDL must take into consideration the human element, and it should assist the individual in understanding the design. Therefore, the ability to express design ideas in a highly verbal, English-like manner is a quality to be desired in a PDL. Based upon their experiences, the authors state unequivocally that Ada has a highly readable syntax. Ada offers a great deal of versatility in terms of readable syntax, highly descriptive identifiers, and user-defined types. Identifiers such as "TRACK\_REPORT\_FROM\_GLOBAL\_COORDINATE\_GENERATION" and "SELECT\_TARGETS\_FOR\_DETECTION\_PROCESSING" exemplify Ada's capability in this regard. Nevertheless, almost all allow for some kind of comment entry. Consequently, liberal use of comment entries and descriptive identifiers, combined with the inherent readability of Ada syntax, can render a design as readable as the program designer cares for it to be.

##### A PDL Should Be Standardized

Standardization does present a problem with most high-order languages. Installation-dependent variations for languages are commonplace. Ada standardization has always been an extremely important language goal. The language reference manual<sup>8</sup> has been submitted to the American National Standards Institute (ANSI) for adoption as a standard. ANSI certification has now been achieved. Standardization of the language will undoubtedly prove to be a powerful force in widespread Ada acceptance and use.

##### A PDL Should Permit Global Data Structures and Subprograms

The Ada programming language's package structure, combined with its visibility rules, makes it simple to declare certain groups of data and/or subprograms as global or limited global [i.e., accessible by some modules, but not all (Figure 4)]. Through the use of the Ada with statement, packages can be made accessible to however many modules the program designer wishes.

as a tutorial on the Ada language. Consequently, the authors feel that the reader should take the initiative to research the Ada language (by examination of any of a number of worthwhile books on the subject<sup>8-10</sup>) in the event that any serious questions should arise.

## PROGRAM DESIGN LANGUAGES

### IDEALIZED PDL

The answer to the question of whether or not Ada can be used as a PDL hinges upon two factors: first, the qualities of an "ideal" PDL; second, the extent to which Ada possess these qualities. The first quality concerning the use of Ada as a PDL involves the dichotomy between "inner" and "outer" syntax.<sup>11</sup> Outer syntax refers to a specific set of syntax rules intended to represent the general structure of software design (e.g., data bases, routines, and access paths), as well as flow of control within routines. Inner syntax has little or no syntactic or semantic restraints and is used to describe (either in general or detailed form) data structures and/or algorithmic features. With regard to the use of Ada as a PDL, the outer syntax would pertain to the actual syntax of Ada. The inner syntax, on the other hand, being less structured, would correspond to Ada comment entries and descriptive identifiers.

The following is a list of what the authors feel are additional significant qualities needed in a PDL in order to ensure its usefulness as a design specification tool. These qualities are an amalgamation of the authors' opinions and the results of the first meeting of the IEEE task group on recommended practices and guidelines for an Ada PDL.<sup>12</sup> The reader is encouraged to add to or delete from this list to suit a particular situation. Ideally, a PDL should

1. Be readable
2. Be standardized
3. Permit global data structures and subprograms
4. Support detail-hiding and data abstraction
5. Permit expression of unspecified parts of the design
6. Permit expression of the physical software system architecture
7. Support top-down design and step-wise refinement
8. Support low-level and high-level sequencing logic
9. Be functionally similar to the implementation language
10. Have unexacting syntax constructs
11. Support validation and verification

When the algorithms are well understood and reliable resource consumption predictions can be made, the above-mentioned approach may be a particularly effective, cost-saving expedient. In such instances, validation of the program structure and of the control flow and event sequencing becomes the important goal. Stages one and two serve this purpose admirably, giving one the best of both worlds. Important intermediate validation milestones are accomplished, but the code developed thereby, also becomes a part of the final system. Ada's contribution to this process is the provision of a tool sufficiently human-engineered and expressive to serve the needs of the system architect, designer, and also the programmers.

The methodology is especially well suited to the real-time programming environment, as can be seen from consideration of the following sequence of events. The program designer, in keeping with the top-down philosophy, proposes some incremental change to the design. The change is interpreted as high-level code, at whatever level of detail is appropriate for the particular stage of design development. The design is then compiled and executed (it is, after all, a bona-fide program). Since the design change has been committed to code and since it is then executed in its natural real-time environment, interactions between the new segment and other program segments can be observed and evaluated. The program designer can then render a judgment as to the success of the design change and accordingly determine the next step in the development. This is not to say that the methodology cannot be used in a non-real-time environment; however, it is fair to say that the methodology is most effective when applied to real-time problems.

With this methodology, each stage in the prototype's development can be saved. By so doing, the traceability of the requirements to the physical program is more easily achieved. Furthermore, it appears that the process described herein can be combined with other methodological concepts to tie the design process more closely to other aspects of software development. Concerning documentation, the design's executability renders it amenable to automated processing for design information (this will be mentioned again later). Concerning program correctness and verification, the concept of program proof-of-correctness techniques<sup>7</sup> appears to be a natural bridge between the second and third stages of prototype development. The authors hope to explore these areas in subsequent research.

In order to determine the validity of this methodology, it will be shown that:

1. The high-level language being used (in this case Ada) is a fitting substitute for a PDL. This will be done by proposing a number of qualities that are indicative of an "ideal" PDL, and by subsequently showing that Ada possesses most, if not all, of these qualities.
2. The computer program produced as a result of this methodology is a valid prototype. Characteristics of an "ideal" prototype computer program will be proposed; the methodology's resultant program will be measured against this model.
3. The authors will elaborate on the development stages by means of illustrative examples.

Numerous references to the Ada programming language syntax are made throughout this report. While the authors have tried to include concise explanations with each syntactic reference, the reader must bear in mind that this report is not intended

Ideally, a prototype computer program should

1. Reconcile functional requirements with physical design
2. Permit evaluation of resource utilization
3. Permit evaluation of control flow and task synchronization
4. Permit evaluation of data flow
5. Permit evaluation of functionality and operational suitability

#### DESIGN PROTOTYPING VS CONVENTIONAL SOFTWARE DEVELOPMENT

What follows is an elaboration on the above qualities of prototype programs. In order to pursue an evaluation of design prototypes, each of the "ideal prototype" criteria will be examined and an assessment of the means for achieving suitable performance in each area will be made.

#### A Prototype Computer Program Should Reconcile Functional Requirements With Physical Design

In order to prevent any misunderstanding on the part of the reader, the authors state unequivocally their interpretations of some commonly heard, though ill-defined, terms. Functional requirements are the capabilities that the program must exhibit to perform its intended job. In a very real sense, the requirements define the job that is to be done. The "functional design" or "logical design" is an expression of the interrelationships between the functional requirements. More specifically, the functional design is composed of four parts: the program's required functional capabilities (i.e., it includes the tasks the program is expected to perform); the relationships among the functions (i.e., which functions use data from other functions and which functions supply data to other functions); the data requirements; and the operator interface with each of the functional areas.

The physical design of a computer program should be closely related to the functional requirements. The functional design of a computer program should map into the physical design and that mapping amounts to the designer's attempt at an implementation that fulfills the functional requirements. Physical modules are instantiated that purportedly perform the functions as specified in the program requirements.

Reconciliation of the physical design and functional requirements is left to the discretion of the designer and/or programmer. The interpretation of logical functional areas into physical ones is not necessarily done on a one-to-one basis. That is, two or more functions may have their processing done in a single physical module, while at the same time two or more physical modules may perform processing for a given functional area.

Traditionally, this reconciliation of the physical with the functional is accomplished by the translation of the functional design into some physical design medium (either a flowchart or a conventional PDL). If incremental development is

practiced at all, it is done by means of computer program "builds." This process may seem at first glance to be closely related to the authors' methodology; however, while there are similarities, the differences between the two procedures are substantial. The reader is asked to consider Table 1. The "build" process essentially implements the table column by column, with groups of columns representing successive builds (i.e., a particular requirement is specified, and each module of the program is implemented so that it supports that particular requirement). Tests and validation follow each build.

TABLE 1. PHYSICAL DESIGN VS FUNCTIONAL REQUIREMENTS MATRIX

FUNCTION MODULE	Function No. 1	Function No. 2	Function No. 3	Function No. 4	Function No. 5	...	Function No. N
MODULE NO. 1	X	X					
MODULE NO. 2	X						
MODULE NO. 3	X		X				
MODULE NO. 4				X	X		
MODULE NO. 5				X			
...							
MODULE NO. M							X

A significant question to ask is whether there are any deficiencies in this approach and how it differs from the prototyping approach described herein. One possible deficiency may be that, with each build, modules can require reworking. It is entirely possible that in the process of modifying the modules, changes occur that invalidate earlier (correct) segments and, consequently, any testing performed on them. Obviously, a row-by-row implementation is desirable to avoid this problem.

The design prototyping methodology, unlike the builds process, emphasizes the physical rather than the functional implementation. Thus, all modules first appear physically as stubs before any of them are developed in more detail. The step-wise refinement process enables each module to be developed in just this way. Once the prototype is developed and validated, specification development can be completed and production of the final system can begin. To avoid the problems mentioned with the build process above, the final system should implement Table 1 on a row-by-row basis.

It is assumed that the logical design and associated requirements are well defined prior to the application of the authors' techniques. The creation of the physical design proceeds in much the same fashion as in the traditional approach.

The program designer interprets the program requirements into some design medium (namely, a high-level programming language) and tests this design against the program's requirements. The major difference occurs in the fact that the physical design is executable (i.e., the logic flow can be traced or simulated) from the very beginning when using the authors' methodology.

#### A Prototype Computer Program Should Permit Evaluation of Resource Utilization

The term resource refers to a great many aspects of a computer system. Most readers will probably think of hardware items (e.g., CPUs, I/O devices, and memory) in connection with computer system resources. In terms of software, a resource can include such items as shared programs, shared data, and messages transmitted among processes. Allocation involves the matching of available resources to the requests for these resources in such a way as to ensure the best design possible.

Validation of resource utilization can be thought of as determining that a given resource is being used so that no conflicts are produced with regard to the use of that resource. This implies that any software resource must first be defined if its allocation is to be made free of conflicts. Likewise, component hardware units must be specified before their allocation can be dealt with. CPU time is a factor in resource allocation that must be addressed. The amount of time a module gets and whether or not it obtains that time when needed are important considerations. Likewise, memory consumption must be considered.

Time constraints can be taken into account by the use of a subprogram designed to stall for a specified period of time (Figure 11). Calls to such a stall routine can be flagged with special comment statements to permit subsequent removal by an APSE tool. Memory can be consumed in a similar fashion if hardware characteristics permit.

```

procedure STALL (DELAY_TIME: in INTEGER) is
  INDEX: INTEGER;
begin
  for INDEX in 1..DELAY_TIME loop
    -- Perform some operation chosen for
    -- its consumption of one unit of time
  end loop;
end STALL;

STALL (20); -- stall for 20 units of time

```

FIGURE 11. CPU TIME CONSUMPTION EXAMPLE

Since hardware resources can affect the allocation of software resources, the failure to specify them until all software resources have been specified can produce serious difficulties. For example, dynamic memory consumption must be considered

along with static module memory requirements. Memory and CPU utilization can be affected by the choice of machine. Machine-dependent hardware considerations include the existence of cache memory, instruction-related (e.g., base register) machine dependencies, and data memory allocations.

Since the authors' methodology requires an executable design, the specification of hardware resources takes place in the same time frame with software resource specification. This enables all resources (hardware and software) to interact from the design's beginning.

#### A Prototype Computer Program Should Permit Evaluation of Control Flow and Task Synchronization

What is control flow? First of all, control has to do with those qualities of a computer program that determine which events will take place, how they will take place, and when they will take place. Control flow is the sequence (in time) in which these events occur in the execution of the computer program.

Therefore, validation of control flow requires the determination that portions of the program execute in the order prescribed, either explicitly or implicitly, in the system requirements. Time sequencing checks need to be performed, based on test scenarios specifically designed for this purpose.

Traditionally, this can only be done upon completion of the design phase. Upon completion of a paper design, the control structure has already been specified; the time sequencing of this control structure is a fundamental part of the design. The interdependence of events in the control structure means that any corrections to the control flow have potentially far-reaching consequences.

In the authors' methodology, validation occurs throughout the design phase, as each component of the control structure is implemented. The component is integrated into the time sequencing scheme; this sequencing is then validated, in spite of the fact that not all components are as yet fully elaborated. Given an Ada run-time support environment, the effects of inserting a new task can, in fact, be tested. In addition, Ada provides the capability of assigning priorities to tasks, which facilitates the testing process. Since the design is executable from its very beginning, control flow test scenarios can be constructed with ever-increasing complexity and can be run at various stages in the design development.

Since functionality is not present to provide specific data values to govern control flow (e.g., to evaluate boolean expressions), conventions must be adopted early in the prototype design and development in order to permit program execution to take place (Figure 12). One such convention is provided for by the Ada separate construct.<sup>8</sup> Such conventions can be based on main path considerations, statistical considerations (i.e., modeling and simulation), or other a priori information. Later, they will be replaced by actual decisions as the design is elaborated.

What is task synchronization? Task synchronization involves a specific kind of task communication. Specifically, receipt of a message (possibly null) by a task occurs in the same time frame as the transmittal of that message by the sending task. Thus, execution of the two tasks is synchronized at that point in time. The two tasks may be said to rendezvous. The Ada task rendezvous mechanism is designed

for the synchronization of concurrent processes, especially when used with the active/passive task constructs (Figure 13). This is the opposite of asynchronous communication, in which a message is received at a time later than when it was sent. The sending task proceeds independent of message receipt by the second task.

```

task body GUN_WEAPON_SYSTEM_INTERFACE is
    .
    .
    .
    function THIS_TARGET_IS_TO_BE_ENGAGED is separate;
    .
    .
    .
begin
    .
    .
    .
    if THIS_TARGET_IS_TO_BE_ENGAGED( ) then
        PERFORM_PERIODIC_UPDATE;
    end if;
    .
    .
    .
end GUN_WEAPON_SYSTEM_INTERFACE;
-----
separate (GUN_WEAPON_SYSTEM_INTERFACE)
function THIS_TARGET_IS_TO_BE_ENGAGED return BOOLEAN;
    VALUE: BOOLEAN := true;
begin
    -- Since the algorithm for the determination of this
    -- boolean condition does not yet exist, a value of
    -- "true" will be returned for simulation purposes
    return VALUE;
end THIS_TARGET_IS_TO_BE_ENGAGED;

```

FIGURE 12. ADA SEPARATE FEATURE

Verification of task synchronization essentially involves the determination that the above-mentioned rendezvous occur at the correct time, in the correct order, and between the appropriate tasks.

Once again, the traditional paper design postpones such a verification until the end of the design phase. "Fine-tuning" the task synchronization structure at this point in the program's development is difficult, because of the interdependence of the tasks.

**Ada Task with Active Structure:**

```

task TRANSMITTER;
task body TRANSMITTER is
begin
  loop
    select
      RECEIVER.GET__MESSAGE;
      -- Initiate rendezvous
    else
      -- If rendezvous not immediately possible,
      -- do something else
    end select;
  end loop;
end TRANSMITTER;

```

**Ada Task with Passive Structure:**

```

task RECEIVER is
  entry GET__MESSAGE;
end RECEIVER;

task body RECEIVER is
begin
  loop
    select
      when CONDITIONS__ARE__APPROPRIATE =>
        accept GET__MESSAGE do
          -- Assimilate message
        end;
      or
        terminate;
    end select;
    -- perform other processing
  end loop;
end RECEIVER;

```

FIGURE 13. ADA TASK SYNCHRONIZATION MECHANISM

With the authors' methodology, all tasks are created and interfaced at the same time, so that any particular task's synchronization with other tasks is verified and, in effect, integrated into the system. Synchronous communication between tasks is directly supported by the Ada rendezvous mechanism (Figure 13), which enables the verification of task synchronization throughout the design phase before costly errors develop. Asynchronous communication requires some form of mailbox.

Ada's suitability in the issue of task synchronization can best be demonstrated by a comparison of Ada's capabilities with those of similar, proven systems. Table 2 exhibits such a comparison of Ada with the AEGIS Tactical Executive System (ATES)<sup>19</sup> and the Digital Equipment Corporation (DEC) VAX Virtual Memory Operating System (VMS).<sup>20,21</sup>

#### A Prototype Computer Program Should Permit Evaluation of Data Flow

Stated simply, data flow concerns how the various modules of the program transmit information to the proper recipients and how these same modules receive information from the proper sources. Given the subtleties of shared data and synchronous/asynchronous message transmittal, this may be an oversimplification. Nevertheless, transferral of information from point A to point B is the essence of data flow. Confirmation that this transferral occurs as specified in the system's requirements constitutes data flow validation.

More specifically, validation of data flow concerns determining if a unit of information is emanating from the correct source and if that information is arriving at its appropriate destinations (Figure 14). It also deals with whether or not information is received in a timely manner (while the information is still "fresh"). Traditionally, this is done upon completion of the design development and is done largely by means of test scenarios.

In a conventional "paper design," shared programs and data can be specified in the form of flowcharts, data tables, or other symbolic representations. Similarly, message traffic can be outlined through the use of message tables. However, such a specification is far more abstract than the programs, data structures, and message networks that these symbols are meant to represent. It is, at best, difficult to confirm the conflict-free use of these software resources based upon a conventional design specification document. Similarly, CPU resource utilization is difficult to evaluate on paper. Rather, validation of resource allocation is normally left to the code and debug phase, when all design components have already been specified. The problems inherent in this approach should be obvious to the reader. By the time the design phase is completed and the specification of software resources has already been made, the relationships of one resource to another are already fixed. As a consequence, the resources' allocations are implicitly specified in the program design. Verification of conflict-free use of resources can truly take place only once those resources have been given substance in a prototype program. Problems with resource allocation that arise this late in the program's development could have been caught sooner if there had been a verification mechanism available early in the design phase.

TABLE 2. A COMPARISON OF ADA, ATES, AND VMS

		Ada	ATES	VMS
I N T E R - M O D U L E  C O M M U N I C A T I O N	Invocation of message by sender	<u>Entry call</u> state- ment in sender	Executive service request from sender	Sender queues write request to receiver's mailbox
	Location of message	Parameter in <u>entry call</u> statement; data structure	User common data; temporary storage	Mailbox message block
	Receiver identification	<u>Entry call</u> statement	Message processing table	Association with mailbox into which message is put
	Acquisition of message	Parameter in <u>accept</u> statement	Pointer to message location	Message placed in receiver's mailbox
	Message receipt method	<u>Accept</u> statement in receiver	Scheduling of receiving module at message entrance	Receiver reads mes- sages in mailbox by queuing read request
	Message receipt acknowledgment	Rendezvous comple- tion; <u>out</u> parameter of accept statement	No acknowledgment	Event flag or soft- ware simulated in- terrupt (optional)
	Sender's earliest time of re-activation	Upon rendezvous completion	Upon receiver's in- sertion in Priority Scheduling Queue	Once the write operation has been initiated
S C H E D U L I N G / S U S P E N S I O N	Initiation of scheduling	Upon <u>task</u> initiation	Module itself; an- other module; ATES program component	System scheduling routines
	Module priority	Optional	All modules priori- tized (by user)	All modules prioritized
	Identification of module to be scheduled	Highest priority <u>task</u> (when applicable)	Contained in scheduling "packet"	Highest priority module that is executable
	Event-driven scheduling	Yes	Yes	Yes
	Module suspen- sion criterion	While <u>entry call</u> pending; while <u>accept</u> pending	Priority-based	Module itself/other module invokes sys- tem service routine
	Inhibition of suspension by module	No	Yes	No

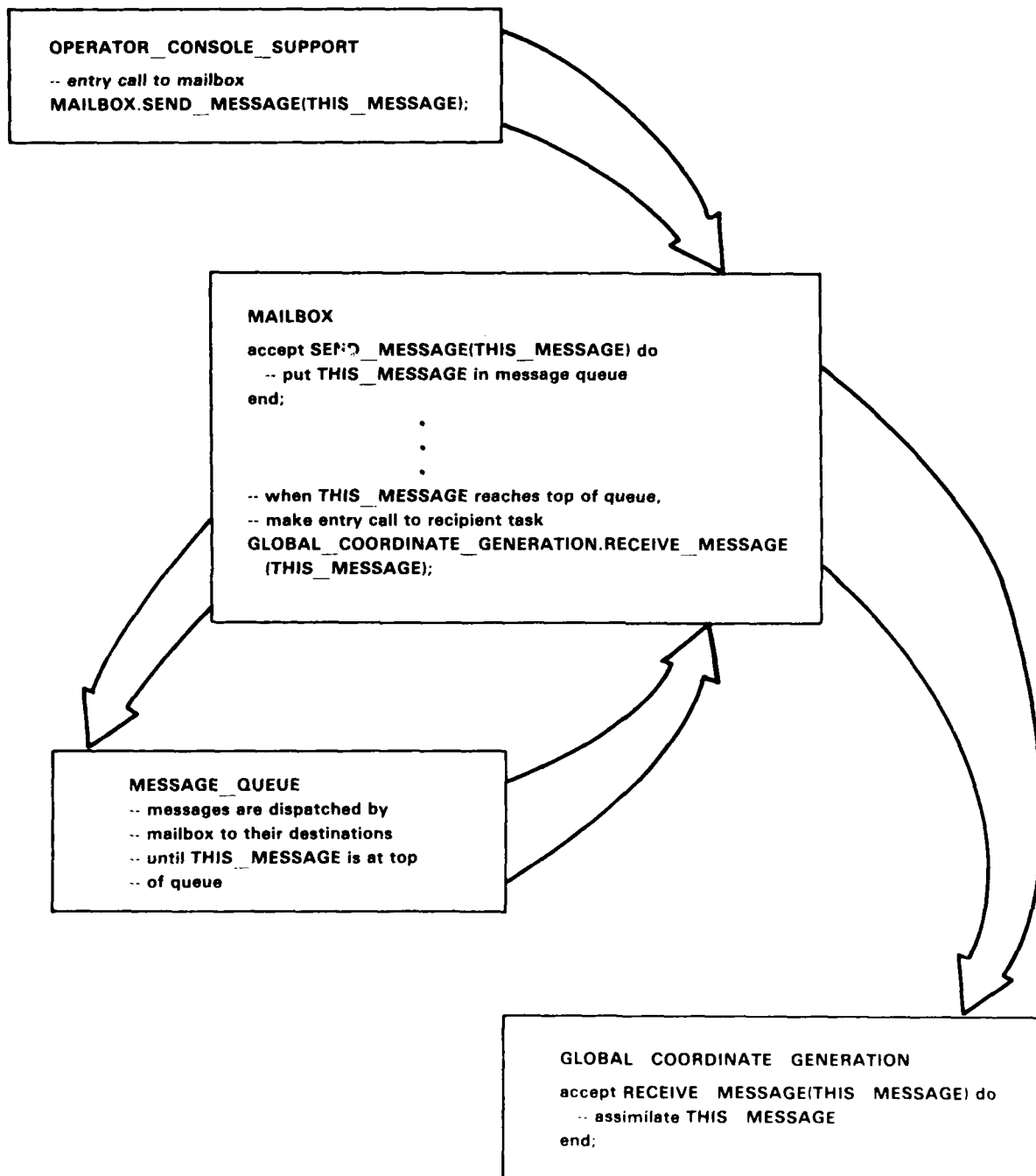


FIGURE 14. INTER-TASK MAILBOX COMMUNICATION STRUCTURE

Such a mechanism is provided by the authors' methodology. Because their methodology enables early prototyping, software resources take on a more tangible quality early in the design phase. For example, allocation of a shared data resource (Figure 15) is far easier if that resource takes the form of an accessible data record in a fledgling prototype than when the resource is expressed as a table in a design specification document. Likewise, utilization of a shared hardware resource can be controlled by means of a binary semaphore (Figure 10). Resource allocation should evolve along with the design, in an orderly fashion, so that the verification process is not left until the entire design has been specified.

```

task PROTECTED_VARIABLE22 is
  entry READ (CURRENT_VALUE: out CHARACTER);
  entry WRITE (NEW_VALUE: in CHARACTER);
end PROTECTED_VARIABLE;

task body PROTECTED_VARIABLE is
  VARIABLE: CHARACTER;
begin
  loop
    select
      accept READ (CURRENT_VALUE: out CHARACTER) do
        CURRENT_VALUE := VARIABLE;
      end;
    or
      accept WRITE (NEW_VALUE: in CHARACTER) do
        VARIABLE := NEW_VALUE;
      end;
    end select;
  end loop;
end PROTECTED_VARIABLE;

```

FIGURE 15. ADA SHARED RESOURCE EXAMPLE

With the authors' methodology, data links between functions are set up early in the design development. This communication network can be validated even before the functions themselves or the messages that are passed back and forth have any substance. (This is especially true for concurrent processing situations.) Furthermore, test scenarios geared toward individual functional areas can be run at various stages in the development of the design. If the code is properly instrumented, flow of data can be verified. This prevents the unnecessary complication of testing the data flow of a function only after its design is completely specified.

The authors' approach has fewer opportunities for error. Detailed functional designs can be integrated into the program on a gradual basis; data flow validation can be done on a similar gradual basis. As different functional areas are elaborated in the program, their data flow can be validated with the other functional areas already developed. (This is completely in keeping with the top-down and step-wise refinement philosophies.)

#### A Prototype Computer Program Should Permit Evaluation of Functionality and Operational Suitability

What constitutes functionality? The quality of functionality involves the program's ability to behave in a manner consistent with the program's functional requirements. The only way to ensure that the program is consistent with its requirements is to execute a series of thorough test scenarios. If the traditional approach is followed, any scenarios designed to test functionality do not take place until the program's design has been completely specified. A paper design remains untried until it is finished.

In the authors' methodology, evaluation of a program design's functionality is made at each stage in the refinement process. Initially, high-level control structures and sequencing logic are tested for compliance with the system requirements. Later, functional areas are tested individually for consistency at the time when their individual designs are "fleshed out." As was said earlier, step-wise refinement of the prototype is part and parcel of the authors' methodology. Therefore, as each function in its turn is developed and matured, its consistency with that particular function's requirements is checked before the next step in the design is undertaken. Evaluation of the finished prototype's functionality is an evolutionary process, with an assessment of functionality being made at the various stages in the program's development.

What exactly constitutes operational suitability in a computer program design? Operational suitability concerns the user's interface with the program. The user, after all, will interact with the finished program on a regular basis and, unless the user is satisfied with the prototype and the final system, the designer has technically failed. In a more definitive sense, the prototype is operationally suitable if, and only if, it adequately performs its intended functions in a manner appropriate from the user's standpoint. The intended functions of the prototype should be specifically identified in the program performance specification (i.e., during the requirements definition phase), and there should be no ambiguities or anomalies in this specification. Similarly, the definition of adequate performance (particularly with regard to the operator interface) should be explicitly stipulated in the performance specifications. Evaluation of the operational suitability of a prototype computer program's design involves measuring it against the specified operational requirements. If the prototype, at a particular stage, meets these requirements (to the user's satisfaction), then the design may be deemed operationally suitable for that level of development. Operational suitability may also involve adequacy of system performance in a suitable environment.

When following the authors' methodology, prototype development provides continuous feedback to the user throughout the design process. The user is then able to monitor the program's maturation process and can interact with the program designer to deal with problem areas as they arise. A significant by-product of this process

is that the user gradually develops an intimate familiarity with the prototype. A detailed understanding of how the prototype works can make the user better equipped to render a more intelligent judgment as to the quality of the design.

## SUMMARY

The prototype produced by the authors' methodology provides a design validation mechanism not readily available with traditional design techniques. While the use of a high-level language as a PDL produces virtually the exact same design as would be generated using a "standard" PDL, the ability to execute the program design throughout the design process is a distinct advantage. A design that executes from its earliest inception certainly has greater credibility and is less likely to contain errors than one that is only scrutinized for executability after the program's design has been specified in its entirety.

## APPLICATION OF THE DESIGN PROTOTYPING METHODOLOGY

### APPLICATION ENVIRONMENT

In this section, the authors have undertaken to describe the design prototyping methodology as it applies to real-time embedded software. The four stages in the development of a physical design are elaborated.

The authors' examples and the program architecture implicit therein are a function of how Ada was used in the context of a trial design effort. These examples in no way imply that this is the only context in which Ada can be used, or that other program architectures are not equally valid. The Ada compiler used by the authors was developed by Telesoft. Inevitably, some compromises and unverified techniques have resulted due to the currently incomplete status of the implementation.

### PHASE ONE OF PHYSICAL DESIGN

The first step in the development of this stage is the identification of system functions and data flow. Subsequently, the determination of formats for typical system modules must be made. The conceptual notion of an Ada task closely parallels that of a module. Furthermore, the ability to execute in parallel (or concurrently, given a single-processor machine) is a functional requirement of the application area. Since concurrent execution is a significant advantage of the Ada task structure, the authors assume that all modules are to be implemented as tasks (Figures 8 and 16). This assumption is generally consistent with the intention stated earlier to develop this methodology within the context of real-time programming.

Given that all modules are implemented as Ada tasks, the question of inter-module communication has to be dealt with. Originally, synchronous module communication was the authors' goal. Unfortunately, the authors' preliminary Ada compiler did not have the conditional entry call feature. Deadlock problems associated with task rendezvous arose when synchronous communication was attempted without this critical structure. It was decided that the only viable solution was a modified

```

task body GUN_WEAPON_SYSTEM_INTERFACE is
  INTERVAL: constant DURATION: = 0.01;
  NEXT_TIME: CALENDAR.TIME;
  -- CALENDAR here refers to a predefined library package,
  -- whose function CLOCK returns the current value of TIME
  -----
  -- Other local data declarations
  *
  *
  *
begin
  accept COMMENCE do
    THIS_TASK.ID: = 11;
    THIS_TASK.NAME: = "Gun Weapon System Interface";
    -- Other initialization routines
    *
    *
    *
    NEXT_TIME: = CALENDAR.CLOCK( ) + INTERVAL;
  end;
  loop
    select
      accept RECEIVE_MESSAGE (MESSAGE: in MESSAGE_TYPE) do
        ASSIMILATE_NEW_INFORMATION;
        -- Assimilate new information into data base
      end;
    or
      -- accept other entries and perform processing
      *
      *
      *
    or
      delay NEXT_TIME - CALENDAR.CLOCK( );
      if THIS_TARGET_IS_TO_BE_ENGAGED( ) then
        PERFORM_PERIODIC_UPDATE;
        -- Perform periodic update routines
      end if;
      NEXT_TIME: = NEXT_TIME + INTERVAL;
      -- Generate message(s) to be sent to
      -- the Mailbox for subsequent transmittal
      -- to its (their) recipient(s)
      MAILBOX.SEND_MESSAGE (THIS_MESSAGE);
    else
      null;
    end select;
    exit when END_OF_PROGRAM;
  end loop;
end GUN_WEAPON_SYSTEM_INTERFACE;

```

FIGURE 16. TYPICAL TASK DEVELOPMENT

n of the "mailbox" task configuration (mentioned earlier), which essentially ed an asynchronous communication network (Figure 14). The system architecture ant from this approach is shown diagrammatically in Figure 17. (The reader note that most embedded computer systems involve interaction with external s. This issue can easily be handled by the creation of additional Ada tasks ed as interrupt handlers.) Figure 18 presents the system architecture en- ated in the form of a package specification. Figure 19 is a visibility chy diagram, which represents program architecture from the compiler's point w.

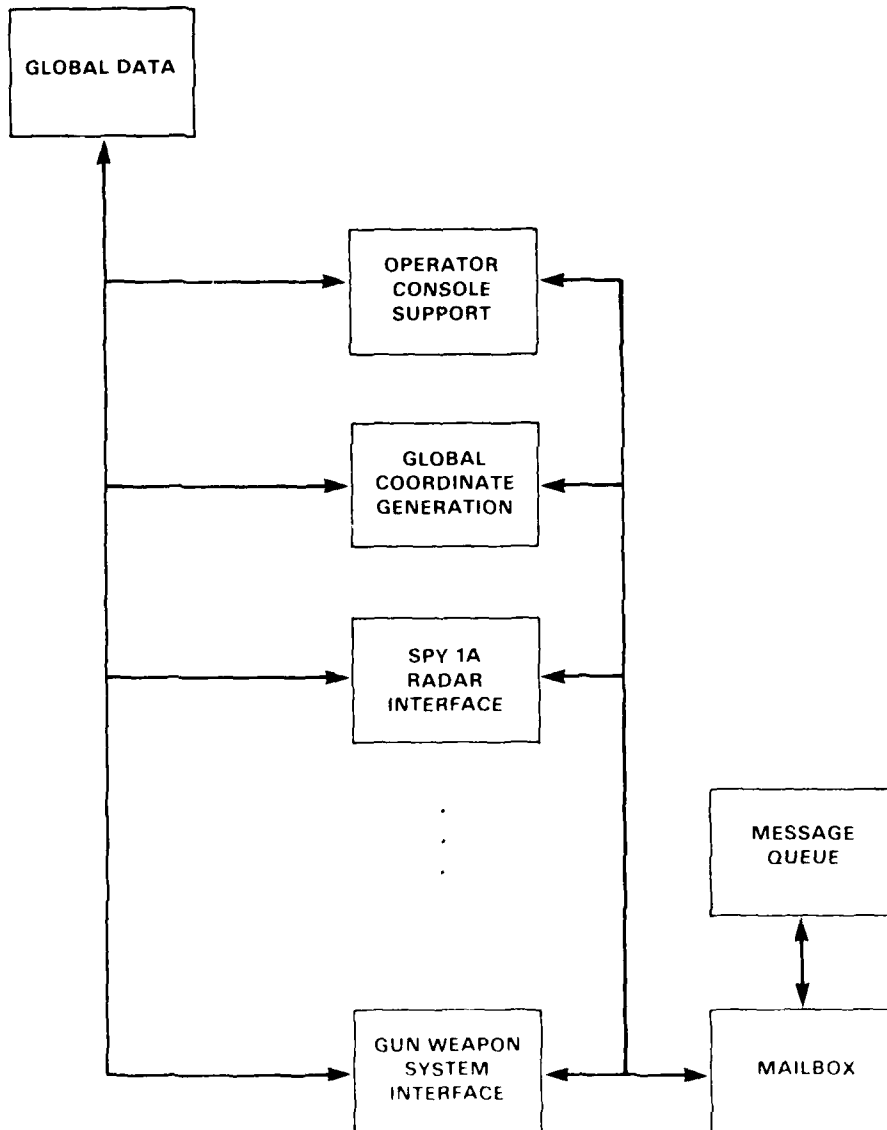


FIGURE 17. SYSTEM MODULE ARCHITECTURE DIAGRAM

```

with GLOBAL_DATA_PACKAGE,
MESSAGE_QUEUE,
MAILBOX_PACKAGE;

package SYSTEM_MODULE_PACKAGE is
  task OPERATOR_CONSOLE_SUPPORT is
    entry RECEIVE_MESSAGE (MESSAGE: in MESSAGE_TYPE);
    -- MAILBOX.SEND_MESSAGE (THIS_MESSAGE);
  end OPERATOR_CONSOLE_SUPPORT;

  task GLOBAL_COORDINATE_GENERATION is
    entry RECEIVE_MESSAGE (MESSAGE: in MESSAGE_TYPE);
    -- MAILBOX.SEND_MESSAGE (THIS_MESSAGE);
  end GLOBAL_COORDINATE_GENERATION;

  task SPY_1A_RADAR_INTERFACE is
    entry RECEIVE_MESSAGE (MESSAGE: in MESSAGE_TYPE);
    -- MAILBOX.SEND_MESSAGE (THIS_MESSAGE);
  end SPY_1A_RADAR_INTERFACE;

  .
  .
  .

  task GUN_WEAPON_SYSTEM_INTERFACE is
    entry RECEIVE_MESSAGE (MESSAGE: in MESSAGE_TYPE);
    -- MAILBOX.SEND_MESSAGE (THIS_MESSAGE);
  end GUN_WEAPON_SYSTEM_INTERFACE;
end SYSTEM_MODULE_PACKAGE;

package body SYSTEM_MODULE_PACKAGE is
  .
  .
  .

end SYSTEM_MODULE_PACKAGE;

```

FIGURE 18. SYSTEM MODULE ARCHITECTURE IN CODE FORM

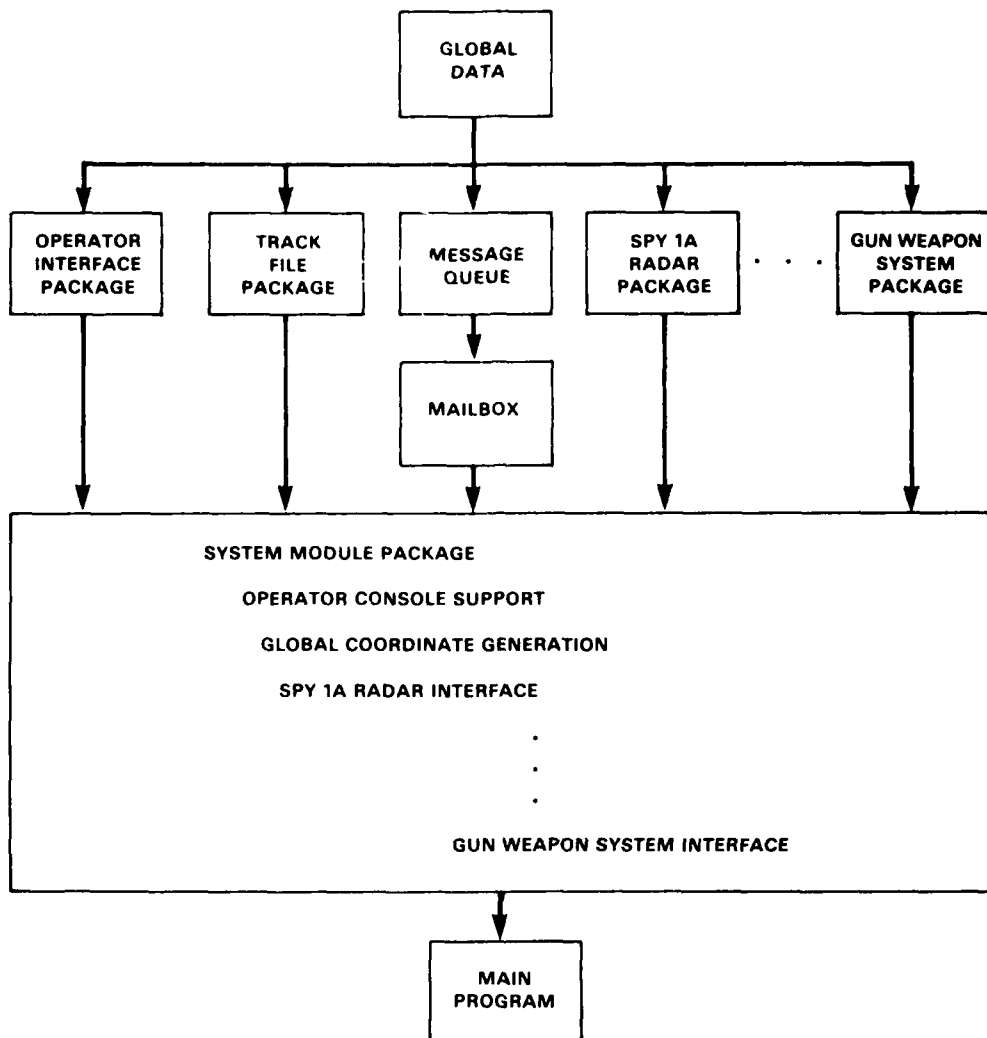


FIGURE 19. VISIBILITY HIERARCHY FROM THE COMPILER'S VIEWPOINT

With the determination of this network, a Master Message List (Figure 20) can be constructed, which consists of a list of stubbed-out messages, each message representing a communication link between two tasks. Each message contains a source and destination, but no content; at this stage of development, only the ability to pass messages is relevant and all resource utilization (e.g., time-driven control loops) is simulated (i.e., the algorithms for the construction of messages, the assignment of conditionals, etc., do not yet exist). Consequently, the development of these resources, at this level of detail, is not based on the actual functionality of the program, but rather on trial design allocations that are to be subsequently verified. Conditions on entry calls and accept statements can be handled much as was described in Figure 12. During phase one, module or task interrelationships are explored and rudimentary subprogram packages are instantiated (Figure 21). Interrelationships among the subprograms are dealt with in phase two.

```

type MESSAGE__TYPE is
  record
    ID: INTEGER;
    SOURCE__NAME: TASK__NAME;
    DESTINATION__NAME: TASK__NAME;
    -- Content field....
    -- to be decoded at destination
  end record;

MASTER__MESSAGE__LIST: array (1..100) of
  MESSAGE__TYPE;

```

FIGURE 20. STUBBED-OUT MESSAGE STRUCTURE

```

with GLOBAL__DATA__PACKAGE;
package GUN__WEAPON__SYSTEM__PACKAGE is
  use GLOBAL__DATA__PACKAGE;

  procedure ASSIMILATE__NEW__INFORMATION;
  procedure PERFORM__PERIODIC__UPDATE;
end GUN__WEAPON__SYSTEM__PACKAGE;

-----

package body GUN__WEAPON__SYSTEM__PACKAGE is
  procedure ASSIMILATE__NEW__INFORMATION is
  begin
    - Assimilate new information
    null;
  end ASSIMILATE__NEW__INFORMATION;

  procedure PERFORM__PERIODIC__UPDATE is
  begin
    -- Perform periodic update
    null;
  end PERFORM__PERIODIC__UPDATE;
end GUN__WEAPON__SYSTEM__PACKAGE;

```

FIGURE 21. TYPICAL SUBPROGRAM PACKAGE AT THE END OF PHASE ONE

The problem of the construction of a generalized message network could possibly be addressed in another, more efficient, manner. The Ada task type capability could be used to create a generalized mailbox with multiple instantiations (one for each system task). The task type would be declared in a generic package, which would enable the mailboxes to handle messages of different types. This multiplicity of mailboxes would still address the problem of message storage and task communication, with the added advantage that not all messages would have to take the same form. Unfortunately, the authors were unable to explore this capability, since their preliminary Ada compiler did not include task types or generics.

Specification of these three structural aspects of the system (modules, message network, and rudimentary messages) amounts to the design of the system-level architecture. Subsequent compilation of the prototype at this stage and its successful execution with a variety of test scenarios complete the test and validation of this phase of the development. The stage is now set for the initiation of phase two.

#### PHASE TWO OF PHYSICAL DESIGN

The second phase of development constitutes what most would regard as a PDL-specified design in the traditional sense. During this phase, the above-mentioned packages of subprograms are developed in greater detail (Figure 22), even though the algorithms themselves do not yet appear. The separate compilation capability of Ada proves to be a distinct advantage in the development of these packages.

Once again, resource allocation is simulated as in phase one. There are, however, some rudimentary functional aspects. Also, conditions are assigned a value so as to validate control flow, even though the algorithmic structure for evaluating those conditions does not yet exist (Figure 12). As was said earlier, this technique essentially predetermines the program's path of execution, but nevertheless permits the evaluation of its flow of control at this stage of development.

Again, the prototype at this stage is compiled; test scenarios apropos to the second level of development are created; and the program is executed, debugged, and reexecuted until performance is satisfactory. In so doing, the prototype is validated for the second phase of its development.

#### PHASES THREE AND FOUR OF PHYSICAL DESIGN

The reader will recall that the authors referred to the third stage of the development as amounting to a degraded version of the finished program. With this idea in mind, the authors singled out certain (critical) modules for extended development. The subprograms associated with these modules are developed in a two-step process: an outline of the subprograms' algorithmic behavior is drawn up in the form of comment entries contained within the subprogram and, subsequently, executable statements are substituted for these comments (Figure 9). Use of program correctness techniques at this point as a bridge between phase two and phases three and four appears to be appropriate.

```

with GLOBAL_DATA_PACKAGE;
package GUN_WEAPON_SYSTEM_PACKAGE is
use GLOBAL_DATA_PACKAGE;

  procedure ASSIMILATE_NEW_INFORMATION;
  procedure PERFORM_PERIODIC_UPDATE;
end GUN_WEAPON_SYSTEM_PACKAGE;

-----

package body GUN_WEAPON_SYSTEM_PACKAGE is
  procedure ASSIMILATE_NEW_INFORMATION is
    procedure GWS_SIMULATION_CONTROL is
      begin
        null;
      end GWS_SIMULATION_CONTROL;

    procedure SHIP_MOTION_GENERATOR_INTERFACE is
      begin
        null;
      end SHIP_MOTION_GENERATOR_INTERFACE;

    procedure TARGET_GENERATOR_INTERFACE;
      begin
        null;
      end TARGET_GENERATOR_INTERFACE;
  begin
    --Assimilate new information into data base
    if MESSAGE.SOURCE_NAME = "Scenario Executive" then
      GWS_SIMULATION_CONTROL;
    elsif MESSAGE.SOURCE_NAME = "Ownship Data generation" then
      SHIP_MOTION_GENERATOR_INTERFACE;
    elsif MESSAGE.SOURCE_NAME = "Ownship Perspective" then
      TARGET_GENERATOR_INTERFACE;
    end if;
  end ASSIMILATE_NEW_INFORMATION;

  .
  .
  .

```

FIGURE 22. TYPICAL SUBPROGRAM PACKAGE DURING PHASE TWO

```

      .
      .
      .
    procedure PERFORM__PERIODIC__UPDATE is
      procedure CALCULATE__BALLISTICS__DATA is
      begin
        null;
      end CALCULATE__BALLISTICS__DATA;
      procedure ISSUE__GUN__ORDERS is
      begin
        null;
      end ISSUE__GUN__ORDERS;
      procedure ASSESS__GUN__EFFECTIVENESS is
      begin
        null;
      end ASSESS__GUN__EFFECTIVENESS;
    begin
      -- Perform periodic update
      CALCULATE__BALLISTICS__DATA;
      ISSUE__GUN__ORDERS;
      ASSESS__GUN__EFFECTIVENESS;
    end PERFORM__PERIODIC__UPDATE;
  end GUN__WEAPON__SYSTEM__PACKAGE;

```

FIGURE 22. TYPICAL SUBPROGRAM PACKAGE DURING PHASE TWO (Continued)

Part of this subprogram development involves a development of the user interface. Not only does the user's ability to exert an influence on the program increase with this stage of development, but feedback information from the program is also generated for the user's enlightenment.

Another enhancement is that many of the messages being exchanged by the tasks are given specific content records during this stage of development.

Compilation, and execution and verification by means of test scenarios once more validate the prototype's competence at this stage.

Suffice it to say that phase four is merely an application of the principles of phase three extended to all system modules and their accompanying subprogram packages for which design validation is considered appropriate. A validated prototype program, designed in accordance with the functional requirements, is the end result of the methodology.

## CONCLUSIONS

## STATE OF THE ART

PDLs based on Ada are currently being developed in industry by, most notably, IBM and TRW. TRW's Ada/PDL consists basically of a "relaxed" version of standard Ada syntax with additional "design narrative" constructs meant to handle unspecified portions of the design.<sup>23</sup> IBM's PDL/Ada is essentially a proper subset of the standard Ada syntax.<sup>24</sup> Since the PDLs are destined to be PDLs in the strict sense, they do not provide executability. However, the IBM version can be analyzed for syntax errors, etc. by an Ada compiler. The authors are encouraged by these efforts, since they foster a program design philosophy not unlike the one described herein. However, it must be pointed out that no matter how closely a PDL is allied with a programming language, it can never offer the validation and verification advantages of an executable design implemented in a true programming language.

## SUMMARY

The advantages of using a high-level programming language as a PDL lie in the ability to construct a prototype program early in the design development phase. It has been shown that most PDL traits can be captured in a high-level language, provided that the program designer is willing to discipline his/her coding techniques to that end. The disadvantages involved include no documentation generation capability with most high-level languages and the fact that most high-level languages do not offer the versatility of Ada. The authors believe that no other high-level programming language to date is better suited to their methodology of design prototyping. Additionally, extensions of this approach upward to encompass automated document generation and downward to encompass program correctness techniques appear to be both beneficial and feasible.

Another advantage of the authors' methodology deals with the quality of the prototype program produced as a result. Because the design is scrutinized every step of the way in its development, the verification process has already been accomplished by the time that the full prototype is implemented. This means that the prototype is substantially error-free. Program correctness techniques potentially can enhance this characteristic to a great degree. Furthermore, since errors in the design are caught at a stage in the design when their correction is relatively simple, the problem of error propagation is greatly diminished. Finally, the transition from conceptual design to tangible program is much smoother than in a conventional system development.

## POSTSCRIPT

In the period since the work in this report was performed, the Institute of Electrical and Electronics Engineering (IEEE) "Ada as a PDL Working Group" has proceeded apace in defining a guideline for use of Ada as a PDL.<sup>25</sup> Note the important distinction in the IEEE effort; the guideline specifically calls for PDL compatibility with the Ada compiler. Any additional design information is to be supplied as special-form comments denoted by "--!" This is, of course, an essential ingredient of the design prototyping methodology in that the language itself represents

the mechanism whereby the design is expressed. A pair of quotes from the current draft guideline serves to illustrate the fundamental philosophical compatibility between the IEEE guidelines and the software design prototyping methodology.

Concerning mainstream design features,

The Ada DL\* should naturally contain features typically associated with a design language; specifically it should be capable of expressing:

- \* Software design structures at various levels of design
- \* Data flow
- \* Execution or control flow
- \* Process or algorithm design
- \* Data definitions
- \* Data usage and references
- \* Interface definition
- \* System element connectivity/dependencies

Concerning support of traceability/allocation features,

The development of software must take into account numerous system requirements. In addition to satisfying functional requirements, the software must be designed to satisfy numerous other complementary requirements, such as performance or security requirements. Many development methodologies utilize a procedure of progressive allocation of these complementary requirements as the development proceeds topdown. Following a design elaboration, each of the functional and complementary requirements are "traced" to those design elements.

It is felt that the allocation procedures and traceability verification generally contribute to quality software developments. For this reason, an Ada DL should support these mechanisms whenever the derivative methodology prescribes them. Below is a candidate list of such complementary requirements to be allocated/traced:

---

\*DL is design language

- \* Performance (Timing and sizing): includes critical timelines, frequencies, capacities, utilizations and limits.
- \* Fault Tolerance: includes error detection, diagnosis, and handling, backup/recovery and reliability redundancy.
- \* Security: includes multilevel security constraints, set/use access restrictions, breach detection and handling.
- \* Distribution: geographical distribution of processing, data storage and access.
- \* Adaption: to provide flexibility for environment operation or user adaption.
- \* Quality: for example those requirements relating to usability, integrity, efficiency, correctness, reliability, maintainability, portability, testability, flexibility.

The authors heartily concur!

## REFERENCES

1. The NAVMAT Software Engineering Environment Working Group, A Software Engineering Environment for the Navy, Naval Material Command (Washington, D.C., 31 March 1982).
2. Headquarters, Department of the Army, A Guide to System Engineering, TM 38-760-1 (Washington, D.C., 30 November 1973).
3. Frederick P. Brooks, Jr., The Mythical Man Month (Reading, Mass.: Addison-Wesley, 1978) pp. 115-123.
4. Lawrence J. Peters, Software Design: Methods & Techniques (New York: Yourdon Press, 1981), p. 16.
5. Robert T. Bevan, Automating the Translation from a Program Design Language (PDL) to Structured Source Code, NSWC TR 80-382 (Dahlgren, Va., November 1980).
6. Don O'Neill, "TRIDENT Integration Engineering," IBM Software Engineering Exchange, Vol. 2 (January 1980), pp. 10-11.
7. David Gries, The Science of Programming (New York: Springer-Verlag, 1981), pp. 1-5.
8. United States Department of Defense, Reference Manual for the Ada Programming Language, Proposed Standard Document (Washington, D.C., July 1980).
9. Henry Ledgard, Ada, an Introduction (New York: Springer-Verlag, 1981).
10. I. C. Pyle, The Ada Programming Language (Englewood Cliffs, N. J.: Prentice-Hall International, 1981).
11. Stephen A. Sutton and Victor R. Basili, FLEX: A Flexible Automated Process Design System, NRL Report 8349 (Washington, D.C., 30 November 1979).
12. The IEEE Working Group on Ada as a PDL, Minutes of the First Meeting (Moorestown, N. J., May 1982).
13. Nathaniel Bowditch, American Practical Navigator, Vol. 2, 1975 edition (Defense Mapping Agency Hydrographic/Topographic Center, 1975), p. 584.
14. A. N. Haberman and Isaac R. Nassi, Efficient Implementation of Ada Tasks, DARPA (Washington, D.C., January 1980).
15. W. Gellert et. al., The VNR Concise Encyclopedia of Mathematics (New York: Van Nostrand Reinhold Company, 1977), pp. 274-275.
16. Renaissance Telesoftware, Inc., The Telesoft ROS Operating System User's Manual, Version 1.0 (5 August 1981).
17. David E. McConnell, Productivity Initiatives for Effective Lifetime Support in the Navy's AEGIS Program (DRAFT), NSWC (Dahlgren, Va., 10 June 1982).

REFERENCES (Continued)

18. Elliot J. Chikofsky, Obtaining Design Information from Existing Software (OXRP/SXRP) (DRAFT), University of Michigan ISDOS Project Technical Memorandum 461 (Ann Arbor, Mich., 17 August 1982).
19. Computer Sciences Corporation, User's Manual for the AEGIS Tactical Executive System (ATES), ACD 1186 (Moorestown, N.J., 8 October 1980).
20. Digital Equipment Corporation, VAX Technical Summary (Maynard, Mass., 1982).
21. Digital Equipment Corporation, VAX/VMS Internals and Data Structures (Maynard, Mass., April 1981).
22. J. D. Ichbiah, et. al., "Rationale for the Design of the Ada Programming Language," ACM SIGPLAN Notices, Vol. 14 (June 1979), p. 11-9.
23. Edward Colbert et. al., A Case for a Simple Ada PDL (DRAFT), TRW Defense Systems Group (Redondo Beach, Calif., March 1982).
24. D. W. Waugh, "Ada as a Design Language," IBM Software Engineering Exchange, Vol. 3 (October 1980), pp. 8-12.
25. The IEEE Working Group on Ada as a PDL, Minutes of the Fourth Meeting (London, England, May 25, 26, 27, 1983).

## DISTRIBUTION

	<u>Copies</u>		<u>Copies</u>
Library of Congress		Commander	
ATTN: Gift and Exchange Division	4	Naval Ocean System Center	
Washington, DC 20540		ATTN: R. Eyres, Code 8302	1
		T. Phillips	1
Naval Joint Project Office		San Diego, CA 92152	
Ballston Tower 2			
Suite 1210		Commanding Officer	
601 North Randolph Street		Naval Air Development Center	
ATTN: LCDR John Kramer	1	ATTN: H. Stuebing	1
Arlington, VA 22203		Warminster, PA 18974	
Headquarters		Commander	
Naval Material Command		Naval Air Systems Command	
Navy Department		Department of the Navy	
ATTN: Owen McOmber, MAT 08Y	1	ATTN: J. Blackmon, PMA 533	1
Washington, DC 20360		B. Zempolich, AIR 360B	1
		Washington, DC 20361	
Commander		Commanding Officer	
Naval Sea Systems Command		Naval Underwater Systems Center	
ATTN: Robert Converse, PMS-408	1	ATTN: T. Conrad, Code 3511	1
CDR S. Kopinitz, PMS-400 B33	1	Newport, RI 02840	
LCDR Langston, SEA 61Y25	1		
LCDR K. Paige, PMS-408	1	Commander	
CDR R. Goodman, PMS-408	1	Naval Electronics Systems Command	
P. Andrews, SEA 61R	1	ATTN: LCDR J. Barnes, ELEX-814	1
R. Vaughn, SEA 003	1	D. Jordan, PME-120-3	1
CAPT Meiniq, PMS-400B	1	J. Machado, ELEX 6134	1
CAPT Lockhart, PMS-400B3	1	M. Potter, ELEX-814	1
CAPT Donegan, PMS-400B5	1	Washington, DC 20360	
CAPT Holloway, SEA 61	1		
CAPT Hatch, SEA 62	1	Commanding Officer	
C. Costanzo, SEA 63B	1	Fleet Combat Direction System	
Washington, DC 20362		Support Activity	
		Dam Neck	
Chief of Naval Operations		ATTN: S. Peele, Code 82	1
ATTN: CDR R. Simpson, OP-942	1	C. Russ	1
CDR A. Hadley, OP-942	1	Virginia Beach, VA 23461	
Washington, DC 20350			

## DISTRIBUTION (Continued)

Copies

Commanding Officer  
 Fleet Combat Direction System  
 Support Activity  
 10 Catalina Boulevard  
 San Diego, CA 92152

1

## Internal Distribution:

1

1

1

1

1

A (T. McKnight)

1

11 (GIDEP)

1

111 (D. Sullivan)

1

131

9

1

0

1

6 (D. Becker)

1

20

1

28 (N. Porter)

1

40

1

1

1

22 (W. Warner)

1

0

1

13 (R. Bevan)

1

30

1

31

1

33 (H. Huber)

1

1

0

1

20

1

20E (D. Green)

1

21

100

24 (E. Dudash, D. McConnell)

2

40

1

402 (R. Cullen)

1

41 (M. Stein)

1

40

1

41 (R. Harrison)

1

42 (E. Price)

1

1

1

40

1

42 (H. Cook)

1

**END**

**FILMED**

7-85

**DTIC**